# TLTracer: Dynamically Detecting Cache Side Channel Attacks with a Timing Loop Tracer

Mingyu Wang*†, Lingjia Meng*†, Fangyu Zheng‡, Jingqiang Lin§, Shijie Jia*, Yuan Ma* and Haoling Fan*†
* Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, CAS, Beijing, China
† School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
‡ School of Cryptology, University of Chinese Academy of Sciences, Beijing, China
§School of Cyber Security, University of Science and Technology of China, Hefei, China

***Abstract*—Recently, cache side-channel attacks have gained increasing attention due to the significant threat they pose to data security. As research advances, these attacks have become more covert and their impact has been widened. To mitigate the threat posed by cache side-channel attacks, numerous detection approaches have been proposed. However, they struggle to capture runtime features or depend heavily on hardware performance counters (HPCs), resulting in a significant number of false negatives or false positives.**

**To address this issue, this paper proposes a broadly applicable runtime feature for identifying cache side-channel attack programs and introduces a dynamic binary analysis approach, TLTracer. TLTracer is runtime trace-based, independent of HPCs and capable of scanning and detecting whether a binary program is malicious before it is deployed in the real world. We implement a prototype of TLTracer and evaluate it with a set of malicious and benign programs. The results show that it can effectively detect the latest cache side-channel attacks without false positives, and offer increased resilience against adversarial evasion compared to other detection tools.**

***Index Terms*—Cache side-channel attacks, Intel Pin, Dynamic binary analysis**

## I. Introduction

Cache side-channel attacks, such as PRIME+PROBE [1], [2], FLUSH+RELOAD [3], take advantage of the shared cache, enable attackers to deduce the victim's sensitive information without privileges by continuously monitoring the cache state. As researchers delve deeper into cache side-channel attacks, these attacks present a significant threat to the security of users' data. The attacker can obtain a victim's sensitive information without accessing it directly, such as key-related data in cryptographic algorithms [4], [3], [5], user activities on Android devices [6], or even the number of distinct items in a user's e-commerce shopping cart [7]. Moreover, cache side-channel attacks are becoming increasingly covert [8], [5] and can evade many detection techniques [9], [10], posing a serious threat to users' information security. Meanwhile, cache side-channel attacks can also be employed for launching transient execution attacks like Meltdown [11] and Spectre [12], further exacerbating the risks.

To mitigate the threats posed by cache side-channel attacks, researchers have proposed numerous solutions aimed at defending against them. New cache designing and memory page coloring requires modifying the OS or hardware [13], [14], making it difficult to flexibly expand and update rapidly. To address these issues, detection solutions focus on swiftly identifying cache side-channel attacks without modification to OS and hardware. These solutions encompass both static and dynamic approaches. Static solutions [15] are usually short of handling the situation where the attackers inject junk bytes. Dynamic approaches commonly rely on monitoring changes in hardware performance counters (HPC) to capture program anomalies [9], [10], [16]. Nevertheless, benign programs running on the system can also trigger fluctuations in HPCs, resulting in a higher false positive rate. Moreover, some approaches necessitate the measurement of threat score [15] or cache state transmission (CST) [16], increasing the burden on users.

In this paper, we concentrate on the runtime behavior patterns associated with cache side-channel attacks. We analyze existing cache side-channel attacks and investigate their common characteristics. According to our investigation, the current cache side-channel attacks have a common runtime behavior pattern: timing a specific microarchitectural event in a loop. Based on this observation, we propose a detection approach, TLTracer, utilizing the common behavior characteristic. TLTracer identifies malicious programs based on dynamic binary analysis and triggers an alarm when necessary.

In practice, TLTracer first captures the execution trace of the program and then extracts the loops within the trace. Finally, taking advantage of the common behavior characteristic, TLTracer analyzes these loops and identifies potential cache side-channel attacks. As a dynamic detection approach, TLTracer liberates itself from dependence on HPCs and makes full use of the runtime feature of binary programs. TLTracer has the ability to effectively identify existing cache side-channel attacks while maintaining a low false positive rate. It is user-friendly as there is no need to compute threat score [15] or measure CST [16]. Meanwhile, it is a flexible tool with a modular design and thus can be readily expanded and updated to cope with potential new cache side-channel attacks in the future.

In summary, this paper presents the following contributions:

- We conduct an analysis of existing cache side-channel attacks, investigating their runtime behavior pattern. We

This work was supported by National Key Research and Development Program of China (No.2022YFB3103301), National Natural Science Foundation of China (No.62272457).

Corresponding author: Fangyu Zheng (Email: zhengfy1028@hotmail.com).

ICC 2024 - IEEE International Conference on Communications | 978-1-7281-9054-9/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICC51166.2024.10622283

introduce the common characteristic, TimingLoop, to identify cache side-channel attacks.
- We design and implement TLTracer, a dynamic binary detection tool for identifying and mitigating cache side-channel attacks. It offers flexibility, user-friendliness, and scalability, facilitating widespread adoption.
- We evaluate TLTracer's detection capability through comprehensive tests, including 10 Proof-of-Concepts (PoCs) of existing attacks. Furthermore, we assess TLTracer's false positive rates using executable files from Linux. The results demonstrate that TLTracer can effectively identify existing cache side-channel attacks without false positives.

## II. Investigating the Common Characteristics of Cache Side-Channel Attacks

In this section, we commence by introducing the prevailing cache side-channel attacks. Subsequently, we perform an in-depth analysis and investigation of their runtime behavior patterns. Finally, we introduce a shared behavioral characteristic termed "TimingLoop".

### A. Cache Side-Channel Attacks

Over the past decades, cache side-channel attacks have posed a great threat to data security. We broadly categorize existing cache side-channel attacks into two main types: FLUSH+RELOAD-type attacks and PRIME+PROBE-type attacks.

**FLUSH+RELOAD-type attacks.** In the situation where the attacker and victim share memory, FLUSH+RELOAD [3] is an efficient and high-resolution method for the attacker. Firstly, in the initialization phase, the attacker utilizes the `clflush` instruction to evict target cache lines from the cache. Subsequently, during the waiting phase, the attacker patiently waits while the victim executes sensitive tasks, potentially altering the cache state. Finally, in the detection phase, the attacker re-accesses the target cache lines and measures access latency. By analyzing this access delay, the attacker is able to determine whether the target data resides in the cache, thereby deducing sensitive data about the victim. In cases where the `clflush` instruction is unavailable, attackers can utilize EVICT+RELOAD [6] as an alternative.

FLUSH+FLUSH [4] is a variant of FLUSH+RELOAD, where attackers use `clflush` to evict the target data during the detection phase and determine whether the target data resides in the cache based on the execution time of `clflush`. FLUSH+FLUSH attacks can bypass some HPCs-based detecting approaches [10], [9], which mainly perceive the presence of attackers based on cache misses.

INVALIDATE+TRANSFER [17] is also a variant of the FLUSH+RELOAD, which leverages the directory protocol for cross-processor attacks without sharing CPU cache. In the detection phase, the attacker accesses the target memory block, analyzing access latency to determine if it's from main memory or other processors. This reveals whether the victim accessed the data, aiding in sensitive information retrieval.

Recently, Guo et al. [8] proposed new attacks: namely PREFETCH+RELOAD and PREFETCH+PREFETCH. The attacker uses the `prefetchw` instruction to initialize the state of the target cache line to "modified". In the detection phase, the attacker measures the latency of reloading or prefetching the target data, and determines whether the victim has accessed the target data based on the latency.

**PRIME+PROBE-type attacks.** PRIME+PROBE-type attacks utilize conflicts on cache access. PRIME+PROBE-type attacks also include three steps: "initialize, wait, and detect". When initiating a PRIME+PROBE attack, the attacker fills the target cache set with their own data (i.e., the eviction set) during the initialization phase. In the detection phase, the attacker accesses the eviction set again and determines whether the victim has accessed the target cache set through access delay.

PRIME+PROBE requires access to the entire cache set for each detection. To enhance efficiency, Purnal et al. [18] propose PRIME+SCOPE, a high-resolution PRIME+PROBE-type attack that is concurrent, persistent, and can be windowless. It only needs to detect one cache line. During the detection phase, the attacker repeatedly accesses the target line to decide whether the target line has been evicted, thereby determining whether the victim has accessed the target cache set. Once initialized, the attacker can only perform the detection step in the loop, and even cancel the waiting phase (i.e., windowless). This greatly improves the resolution of the attack.

RELOAD+REFRESH [5] does not require the eviction of target data from the Last Level Cache (LLC), therefore it is more hidden. During the initialization phase, the attacker arranges access to the eviction set and the target memory block skillfully to avoid the target memory block being evicted from the LLC while making the target data the eviction candidate of the low-level cache. During the detection phase, the attacker measures the latency of accessing the target address to determine if the victim has accessed the target data.

PRIME+ABORT [19] attackers employ the Intel Transaction Synchronization Extensions (TSX) during the detection phase to monitor aborts associated with the access to the target memory block, rather than measuring the time required to access the eviction set.

### B. TimingLoop for Identifying Cache Side-Channel Attacks

In the "initialize-wait-detect" method discussed in Section II-A, attackers can typically only discern one bit at a time. As a result, to gather sufficient information, attackers repetitively execute these three steps.

We investigate and summarize the runtime behavior of existing cache side-channel attacks in Table I, including the initialization and detection methods and other features. According to the introduction in Section II-A, we know that attackers primarily extract sensitive information during the "detect" step. Therefore, the detection phase plays a pivotal and irreplaceable role.

By analyzing Table I, it becomes evident that as research advances, many of the latest cache side-channel attacks no

TABLE I: Cache side-channel attacks and characteristics.

| Attack Method | Initialization Method | Detection Method | Evict Target Data from LLC | Cause Significant Cache Misses | Include TimingLoop |
|---|---|---|---|---|---|
| FLUSH+RELOAD | flush | Timing memory access | ✓ | ✓ | ✓ |
| EVICT+RELOAD | evict | Timing memory access | ✓ | ✓ | ✓ |
| FLUSH+FLUSH | flush | Timing flush | ✓ | ✗ | ✓ |
| INVALIDATE+TRANSFER | flush/evict | Timing memory access | ✓ | ✓ | ✓ |
| PREFETCH+RELOAD | prefetch | Timing memory access | ✗ | ✗ | ✓ |
| PREFETCH+PREFETCH | prefetch | Timing prefetch | ✗ | ✗ | ✓ |
| PRIME+PROBE | prime | Timing memory access | ✓ | ✗ | ✓ |
| PRIME+SCOPE | prime | Timing memory access | ✓ | ✗ | ✓ |
| RELOAD+REFRESH | prime | Timing memory access | ✗ | ✗ | ✓ |
| PRIME+ABORT | prime | Waiting for an abort | ✓ | ✗ | ✗ |

longer result in the eviction of target data from the LLC and do not trigger an abundance of cache misses. Therefore, HPCs-based detection methods [10], [9] are difficult to handle emerging cache side-channel attacks. Our investigation reveals that, except for PRIME+ABORT, these cache side-channel attacks typically involve multiple detection steps. To the best of our knowledge, however, the most recent processors no longer support Intel TSX, rendering PRIME+ABORT attacks unfeasible on these new processors.

Listing 1: A classic core code from [3] to measure cache access latency.

```
static inline uint32_t
    memaccesstime(void *v) {
      uint32_t rv;
      asm volatile (
           "mfence\n"
           "lfence\n"
           "rdtscp\n"
           "mov %%eax, %%esi\n"
           "mov (%1), %%eax\n"
           "rdtscp\n"
           "sub %%esi, %%eax\n"
           : "=&a" (rv): "r" (v): "ecx",
           "edx", "esi");
      return rv;
}
void fr_probe(fr_t fr, uint16_t *results) {
  ......
  int l = vl_len(fr->vl);
  for (int i = 0; i < l; i++)  {
    void *adrs = vl_get(fr->vl, i);
    int res = memaccesstime(adrs);
    ......
   }
}
```

Based on our investigation, we introduce a common runtime behavior characteristic named **TimingLoop**. TimingLoop signifies a loop structure used by attackers to measure the delay of particular microarchitectural events, including memory access, flush, and prefetch. We employ TimingLoop as a distinctive feature for identifying cache side-channel attacks.

Attackers frequently use the `rdtsc/rdtscp` instruction when measuring the latency of microarchitectural events [20]. For the sake of brevity, we will collectively refer to them as the `rdtsc` instruction in the rest of this article. Listing 1 illustrates a classic code snippet for assessing memory access latency. Based on our investigation, recent cache side-channel attacks like PREFETCH+RELOAD, PRIME+SCOPE, as well as some transient execution attacks such as Meltdown [11] and Spectre [12], have all employed similar code for timing microarchitectural events. Consequently, the TimingLoop described in this article takes into account the use of the `rdtsc` instruction for time measurement and leaves the exploration of alternative timing methods for future work.

## III. Design and Implementation

Drawing from our analysis of the behavioral characteristics of cache side-channel attacks, we design TLTracer, a detection tool that utilizes TimingLoop as an identity to detect and determine whether an executable file is a potential cache side-channel attack program. This section introduces the architecture and fundamental components of TLTracer, along with specific functions and implementation details of each component. TLTracer is implemented and evaluated on Ubuntu 18.04 running on the PC with Intel Core i7-6820HQ CPU, equipped with 8GB of memory.

### A. Overview

As illustrated in Figure 1, before release, application store operators must conduct a comprehensive review of binary programs, covering various aspects like quality, performance, security, etc. TLTracer can serve as a valuable component in the security review for examining programs submitted to the app store. Moreover, TLTracer can also be integrated into the kernel or hardware, collecting and processing runtime information of programs, timely discovering potential attack processes and killing them.

The architecture of TLTracer is shown in Figure 1, which consists of three components: Execution Tracker (ET), Loop Profiler (LP), and Decision Maker (DM). The functions of these three components are briefly described as follows:

- ET accepts binary programs as input and documents the details of the program's execution path throughout its runtime. This process results in the generation of a comprehensive trace log file.
- LP first parses the trace log and skillfully extracts loops from the program's execution. Then, LP refines the identified loops through specific filtering strategies, removing loops that cannot be TimingLoops.
- DM performs a comprehensive analysis of these filtered loops. It scrutinizes each loop to discern whether it conforms to the characteristics of a TimingLoop.

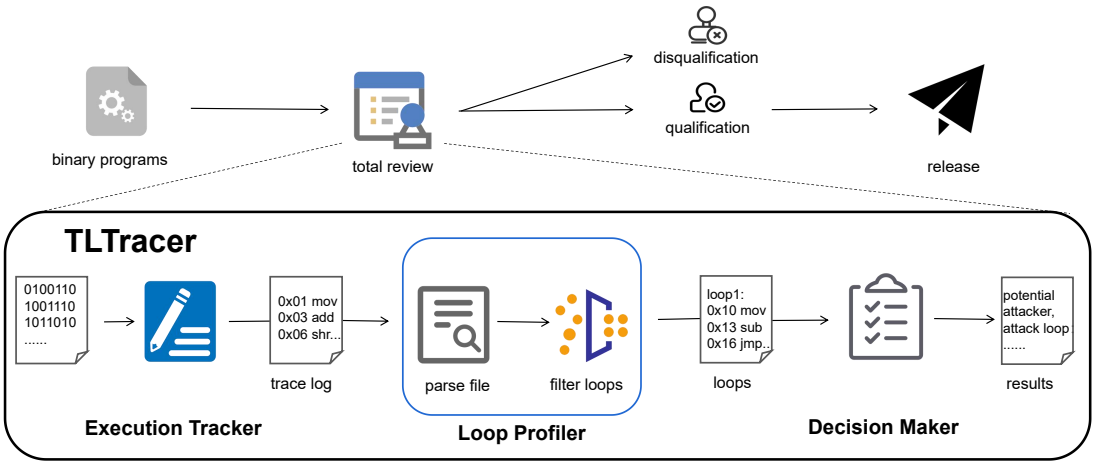


Fig. 1: TLTracer components.

Finally, based on the analysis, TLTracer determines whether the binary program is a potential cache side-channel attack program and generates a conclusive decision report for user reference.

### B. Execution Tracker

Execution Tracker mainly captures the execution trace of a program, and records detailed runtime information, including the address of each instruction and the specific instruction name (opcode). For the jump instructions, ET additionally records the operands of the instruction, as well as the contents of each register and the accessed memory addresses, to assist in subsequent analysis and extraction of loops.

ET is built based on Intel Pin [21], an off-the-shelf dynamic binary instrumentation framework developed by Intel. ET performs instruction-level instrumentation on binary programs, recording the address and opcode (such as `mov`, `rdtsc`, `jmp`, etc.) of each instruction (except for system calls). If ET encounters a jump instruction, including 33 specific jump instructions such as `jmp`, `jz`, `je`, `jc`, etc, it records the operand of the instruction, which may be an immediate value, the name of a register, or a memory address. Therefore, ET also records register information and the accessed memory address. For non-jump instructions, ET only records the address and opcode of the instruction. This not only enhances the efficiency of ET but also alleviates the workload of LP, thereby improving LP's efficiency and conserving memory. Finally, ET records the execution trace in a log file and transfers it to LP.

### C. Loop Profiler

Loop Profiler parses the information recorded by ET and analyzes the log file to extract loops, then filters these loops.

In previous work, CryptoHunt [22] proposed a method for identifying loops from the execution trace of a program. We customize and extend the loop detection algorithm employed in CryptoHunt to align with the specific requirements of TLTracer. First, CryptoHunt was originally designed for the 32-bit operating system. To make it compatible with our 64-bit operating system, we made necessary adjustments to its variants, constants, registers, etc. Second, we introduce `L_num` for each identified loop iteration to record its execution count. Furthermore, we incorporate support for handling indirect jumps, where the operand of the jump instruction is not an immediate value. In detail, when the operand of the jump instruction is a register or a memory address, LP queries the register information or the accessed memory address recorded by ET to determine the target address.

In addition, we adjust the filtering strategy to adapt to our detection approach. The adjusted LP uses the following strategy to filter the identified loops:

1) Filtering out loops with empty loop bodies.
2) Filtering out loops with excessively long bodies. In the case of a TimingLoop, if an attacker inserts a significant amount of instructions, causing the loop body to become too large, the measurement time will inevitably increase. This extended latency may cause the attacker to miss crucial secret bits, reducing effectiveness and, in some cases, leading to failure. Hence, we employ a filter to eliminate loops with excessive sizes. In detail, if the size of a loop body is 0 or greater than 0xffff, LP deletes it.
3) Filtering out loops that do not contain `rdtsc` instructions. Based on our analysis in Section II-B, TimingLoop is characterized by the presence of `rdtsc` instructions. Consequently, LP examines each loop body to verify the presence of `rdtsc` instructions, removing any loop bodies lacking this characteristic.

Finally, LP passes the filtered loop bodies to DM, reduces the workload for DM, and enhances the system's overall efficiency.

### D. Decision Maker

Decision Maker processes the filtered loop bodies to identify TimingLoops and assess the frequency of TimingLoop execution. It is worth noting that there could be conditional branches in a loop body leading to different loop iterations, as explained in CryptoHunt. In order to provide a more detailed analysis of behavioral characteristics, DM analyzes loop iterations instead

of loop bodies. A TimingLoop is also a characteristic at the loop iteration level.

Firstly, DM identifies TimingLoop based on the definition of TimingLoop in Section II-B. Specifically, for each loop iteration, DM checks whether a microarchitectural event was executed between two calls of `rdtsc` instruction. Different microarchitectural events are determined based on opcodes, such as `clflush`, `mov ptr`, `prefetchw`, `prefetchnta`, etc, for flush, prefetch, and memory access respectively. Secondly, we set a threshold (`TL_threshold`) for the number of TimingLoop executions to identify cache side-channel attacks. DM checks the number of TimingLoop executions, `L_num`. If `L_num` is greater than `TL_threshold`, DM identifies the binary program as a potential cache side-channel attack program and timely reports it to the user.

Based on our observation, TimingLoop is common in many benign programs, but it is only executed 1-2 times in benign programs. Compared to benign programs, cache side-channel attack programs need to obtain enough sensitive bits, leading to the core TimingLoop generally being executed hundreds or thousands of times. Therefore, we ultimately set `TL_threshold` to 10. We will provide a detailed introduction to the relevant observations in Section IV.

## IV. Evaluation

In this section, we evaluate TLTracer's performance and effectiveness within the lab environment described in Section III. We assess TLTracer's efficacy using 10 PoCs of existing cache side-channel attacks and examine its false positive rate by testing 56 benign binary programs. Additionally, we record the time required by TLTracer for these tests.

### A. Testing for Malicious Programs

We use Mastik [23] to launch the FLUSH+RELOAD, FLUSH+FLUSH, and PRIME+PROBE (L1 cache and LLC) attacks. We employ publicly available PoC from [8] for testing PREFETCH+PREFETCH, PREFETCH+RELOAD, [18] and [5] for testing PRIME+SCOPE and RELOAD+REFRESH, respectively. Additionally, we employ PoCs from [11] and [12] to carry out Meltdown and Spectre. The results of TLTracer's detection of cache side-channel attack programs are presented in Table II.

From Table II, we can see that PRIME+PROBE (LLC), PRIME+SCOPE, and PREFETCH+RELOAD have larger TimingLoops in comparison to loop bodies, which we attribute to the execution of numerous distinct branches within a loop body. For instance, in PRIME+PROBE, the pointer-chasing technique described in [1] could lead to different runtime loop iterations within the same loop. Additionally, in table II, `L_num` represents the maximum `L_num` among all TimingLoops, indicating the significance of the most prominent TimingLoop. Table II also records parsing time. It's important to clarify that the parsing time actually refers to the offline analysis time, which encompasses the processing time of LP and DM that serve as the primary components responsible for parsing log files, extracting and analyzing loops.

TABLE II: The Results of Testing Malicious Programs

| Attack | Loops bodies | TimingLoops | Parsing Time (sec) | `L_num` |
|---|---|---|---|---|
| FLUSH+RELOAD | 309 | 23 | 43.68 | 499 |
| FLUSH+FLUSH | 287 | 33 | 516.27 | 1998 |
| PRIME+PROBE (L1) | 180 | 23 | 46.13 | 499 |
| PRIME+PROBE (LLC) | 156 | 280 | 32.88 | 72 |
| PREFETCH+PREFETCH | 183 | 93 | 36.49 | 1387 |
| PREFETCH+RELOAD | 190 | 12080 | 71.77 | 174 |
| PRIME+SCOPE | 242 | 13271 | 64.48 | 128 |
| RELOAD+REFRESH | 206 | 22 | 93.21 | 100723 |
| Meltdown | 52 | 4 | 11.53 | 256969 |
| Spectre | 164 | 26 | 31.64 | 98 |

### B. Testing for Benign Programs

When evaluating the false positive rate of TLTracer, we utilized 56 binary executable files that included Linux system tools, programming and development class tools, network tools, document and multimedia processing tools, editors, and cryptographic programs. The cryptographic programs use the OpenSSL 1.1.0h library for cryptographic operations.

TABLE III: The Results of Testing Benign Programs

| Type | Programs | Loop bodies | TimingLoops | Parsing Time (sec) | `L_num` |
|---|---|---|---|---|---|
| System Tools | ls | 312 | 21 | 16.74 | 1 |
| | cat | 196 | 20 | 12.09 | 0 |
| | cp | 249 | 21 | 12.09 | 1 |
| | mv | 244 | 21 | 15.11 | 1 |
| | head | 200 | 20 | 15.23 | 1 |
| | 7z | 233 | 20 | 15.05 | 1 |
| | pwd | 197 | 23 | 15.78 | 1 |
| | rm | 197 | 20 | 16.28 | 1 |
| | mkdir | 241 | 19 | 17.73 | 1 |
| | rmdir | 193 | 20 | 16.73 | 1 |
| | ps | 477 | 19 | 156.55 | 1 |
| | grep | 374 | 21 | 21.31 | 1 |
| | top | 708 | 21 | 82.22 | 1 |
| | sed | 299 | 20 | 19.69 | 1 |
| | awk | 176 | 23 | 12.61 | 1 |
| | touch | 193 | 20 | 13.33 | 1 |
| | df | 439 | 20 | 25.33 | 1 |
| | du | 282 | 19 | 14.25 | 1 |
| | echo | 197 | 20 | 13.12 | 1 |
| | find | 395 | 30 | 20.03 | 1 |
| | kill | 307 | 21 | 19.75 | 1 |
| | more | 319 | 22 | 15.66 | 1 |
| | less | 464 | 36 | 17.28 | 1 |
| | mount | 433 | 20 | 48.12 | 1 |
| | umount | 336 | 20 | 28.14 | 1 |
| | which | 203 | 20 | 14.84 | 1 |
| | whereis | 248 | 20 | 25.42 | 1 |
| Programming | gcc | 503 | 61 | 22.97 | 1 |
| | g++ | 527 | 120 | 22.07 | 1 |
| | make | 486 | 52 | 38.89 | 1 |
| | ld | 652 | 23 | 69.07 | 1 |
| | ldd | 801 | 261 | 46.09 | 1 |
| | as | 465 | 22 | 49.48 | 1 |
| | nm | 433 | 23 | 33.48 | 1 |
| | objdump | 332 | 20 | 37.25 | 1 |
| | strip | 493 | 40 | 34.69 | 1 |
| | yacc | 233 | 20 | 16.81 | 1 |
| | bison | 692 | 19 | 59.62 | 1 |
| | perl | 692 | 19 | 59.62 | 1 |
| Network | wget | 1467 | 949 | 574.8 | 2 |
| | scp | 300 | 24 | 18 | 1 |
| | rsync | 279 | 61 | 20.86 | 1 |
| | ifconfig | 374 | 21 | 18.24 | 1 |
| | netstat | 595 | 65 | 39.72 | 1 |
| | ss | 412 | 23 | 37.8 | 1 |
| | nc | 180 | 23 | 18.07 | 1 |
| Documents and Multimedia | convert | 382 | 39 | 53.51 | 1 |
| | gs | 352 | 54 | 85.7 | 1 |
| | pdftotext | 241 | 23 | 33.48 | 1 |
| Editor | vim | 518 | 20 | 38.86 | 1 |
| | nano | 303 | 21 | 30.39 | 1 |
| Crypto | AES | 199 | 33 | 232 | 1 |
| | ECDSA | 362 | 184 | 546 | 1 |
| | RSA | 401 | 70 | 471 | 1 |
| | MD5 | 201 | 23 | 224 | 1 |
| | SM3 | 225 | 21 | 246 | 1 |

Table III presents TLTracer's results for the detection of benign programs. Similar to Table II, Table III also includes details such as the number of identified loop bodies and TimingLoops, `L_num`, and the recorded execution time. By comparing the results from Table II and III, it becomes evident that benign programs almost contain TimingLoops. However,

they typically execute TimingLoops only once or twice. In contrast, malicious programs execute TimingLoops hundreds or even thousands of times. Hence, we employ a combination of TimingLoop and `L_num` to enhance the accuracy of attacker identification while minimizing false positives. By configuring `TL_threshold` to a value of 10, we observe no false positives among the 56 benign programs and no missed detections among the attack programs. Notably, as indicated in Table III, we find that even with `TL_threshold` set to 2, there were no false positives among the 56 benign programs. But we set `TL_threshold` to 10 to deal with the scenarios in which there might be more frequent instances of TimingLoop in some applications. Subsequently, by utilizing both TimingLoop and `L_num`, DM can readily differentiate between benign and malicious programs, ensuring a low false positive rate.

Furthermore, as evident from the experimental results, we can tune `TL_threshold` flexibly within a relatively wide range while maintaining a low false positive rate to address the possibility of encountering lower values of `L_num` in cache side-channel attack programs in the future.

## V. Related Work

In this section, we offer a succinct overview of the proposed detection schemes for cache side-channel attacks.

**Static Detection Methods.** Static detection methods excel at processing large binary files but lack the ability to monitor a program's runtime behaviors and characteristics. They could be bypassed if attackers inject junk code. MASCAT [15] can detect multiple microarchitectural attacks through binary files. However, MASCAT cannot detect newly emerging attacks, such as PREFETCH+PREFETCH.

**Dynamic Detection Methods.** Dynamic detection methods are capable of capturing runtime features, but they often rely on HPCs and suffer from a high false positive rate. SCAGuard [16] employs Pin and a control flow graph to capture attack-relevant syntactic code information, aiming to detect cache side-channel attacks. However, it faces challenges in practical implementation due to the need to use a cache simulator for CST measurement, a task complicated by substantial noise presented in real-world caches. SCADET [24] is a dynamic tracking and offline analysis method. However, SCADET has an average false positive rate of 7.4% and a narrow detection scope, primarily identifying PRIME+PROBE attacks. Some detection methods [9], [10] rely on monitoring changes in HPCs to capture program anomalies. They face a higher false positive rate because benign programs can also cause HPCs fluctuations.

## VI. Conclusion

In this paper, we propose a method based on dynamic analysis of binary programs to detect cache side-channel attacks. We analyze the existing cache side-channel attacks in detail and propose the TimingLoop as the recognition feature of cache side-channel attack programs. We implement TLTracer and evaluate it comprehensively. The results demonstrate that TLTracer can identify all the cache side-channel attacks without false positives. In summary, TLTracer stands out as a user-friendly, universal, and flexible detection tool. It can be easily deployed in the application stores as a pre-release binary program detection tool.

## References


[1] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks are Practical," in *IEEE S&P*, 2015, pp. 605–622.

[2] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: The Case of AES," in *CT-RSA*, 2006, pp. 1–20.

[3] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *USENIX Security*, 2014, pp. 719–732.

[4] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: A Fast and Stealthy Cache attack," in *DIMVA*, 2016, pp. 279–299.

[5] S. Briongos, P. Malagón, J. M. Moya, and T. Eisenbarth, "RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks," in *USENIX Security*, 2020, pp. 1967–1984.

[6] D. Gruss, R. Spreitzer, and S. Mangard, "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches," in *USENIX Security*, 2015, pp. 897–912.

[7] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-Tenant Side-Channel Attacks in PaaS Clouds," in *ACM SIGSAC*, 2014, pp. 990–1003.

[8] Y. Guo, A. Zigerelli, Y. Zhang, and J. Yang, "Adversarial Prefetch: New Cross-Core Cache Side Channel Attacks," in *IEEE S&P*, 2022, pp. 1458–1473.

[9] M. Payer, "HexPADS: A Platform to Detect "Stealth" Attacks," in *ESSoS*, 2016, pp. 138–154.

[10] M. Chiappetta, E. Savas, and C. Yilmaz, "Real Time Detection of Cache-based Side-Channel Attacks using Hardware Performance Counters," *Appl. Soft Comput.*, vol. 49, pp. 1162–1174, 2016.

[11] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *USENIX Security*, 2018, pp. 973–990.

[12] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *IEEE S&P*, 2019, pp. 1–19.

[13] M. K. Qureshi, "Ceaser: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping," in *MICRO*, 2018, pp. 775–787.

[14] J. Shi, X. Song, H. Chen, and B. Zang, "Limiting Cache-Based Side-Channel in Multi-Tenant Cloud Using Dynamic page coloring," in *IEEE/IFIP DSN Workshops*, 2011, pp. 194–199.

[15] G. Irazoqui, T. Eisenbarth, and B. Sunar, "MASCAT: Preventing Microarchitectural Attacks Before Distribution," in *CODASPY*, 2018, pp. 377–388.

[16] L. Wang, L. Bu, and F. Song, "SCAGuard: Detection and Classification of Cache Side-Channel Attacks via Attack Behavior Modeling and Similarity Comparison," in *DAC*, 2023, pp. 1–6.

[17] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cross processor cache attacks," in *Asia CCS*, 2016, pp. 353–364.

[18] A. Purnal, F. Turan, and I. Verbauwhede, "Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks," in *ACM SIGSAC*, 2021, pp. 2906–2920.

[19] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, "Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX," in *USENIX Security*, 2017, pp. 51–67.

[20] R. Martin, J. Demme, and S. Sethumadhavan, "Timewarp: Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-Channel Attacks," in *ISCA*, 2012, pp. 118–129.

[21] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," *ACM sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[22] D. Xu, J. Ming, and D. Wu, "Cryptographic Function Detection in Obfuscated Binaries via Bit-Precise Symbolic Loop Mapping," *IEEE S&P*, pp. 921–937, 2017.

[23] Y. Yarom, "Mastik: A micro-architectural side-channel toolkit," 2016.

[24] M. Sabbagh, Y. Fei, T. Wahl, and A. A. Ding, "SCADET: A Side-Channel Attack Detection Tool for Tracking Prime-Probe," in *ICCAD*, 2018, pp. 1–8.