SpecScope: Automating Discovery of Exploitable Spectre Gadgets on Black-box Microarchitectures

Najmeh Nazari*‡, Behnam Omidi†‡, Chongzhou Fang*, Hosein Mohammadi Makrani*,
Setareh Rafatirad*, Avesta Sasan*, Houman Homayoun*, and Khaled N. Khasawneh†
* University of California, Davis, Emails: {nnazari,czfang,hmakrani,srafatirad,asasan,,hhomayoun}@ucdavis.edu
† George Mason University, Emails: {bomidi,kkhasawn}@gmu.edu

‡Co-authors contributed to the manuscript equally

***Abstract*—Transient execution attacks pose information leakage risks in current systems. Disabling speculative execution, though mitigating the issue, results in significant performance loss. Accurate identification of vulnerable gadgets is essential for balancing security and performance. However, uncovering all covert channels is challenging due to complex microarchitectural analysis. This paper introduces SpecScope, a framework for automating the detection of Spectre gadgets in code using a black-box microarchitecture approach. SpecScope focuses on contention between transient and non-transient instructions to precisely identify and reduce false-positive Spectre gadgets, minimizing mitigation overhead. Tested on public libraries, SpecScope outperforms existing methods, reducing False-Positive rates by 8.9% and increasing True-Positive rates by 10.4%.**

I. Introduction

CPUs implement the Instruction Set Architecture (ISA) through their microarchitecture, varying in unit count and types, invisible to programmers. This variance can cause delays and contention in instruction execution, risking data leaks [1]. Microarchitectural resources are divided into stateless, like functional units, and stateful, such as caches and Translation Lookaside Buffers. Stateless resources don't retain state, unaffected by previous instructions. In contrast, stateful resources maintain a state affected by instruction execution. Resources altering states during transient execution are susceptible to transient execution attacks, a severe form of microarchitectural attacks [2], [3].

Transient execution, occurring in instances like branch misprediction, executes instructions that are later removed from the CPU pipeline. While ensuring execution correctness, it can leave microarchitectural traces, such as data in the cache, which transient execution attacks exploit. Spectre is a prominent example, using control-flow misprediction to trigger such execution [4].

To counter these attacks, hardware defenses are being developed [5], [6], [7], with CPU vendors addressing Spectre variants in future CPUs. Nonetheless, some microarchitectures remain vulnerable, necessitating software-level defenses. Completely halting speculative execution is effective but costly in performance. A more efficient method involves using serialization instructions to target and patch only vulnerable code segments ('gadgets') [8]. However, this approach primarily detects gadgets in known attack variants, leaving those from unknown causes, like frequency throttling effects [9], at risk.

To identify all contention-based Spectre attacks, it's crucial to find all related covert channels. This requires deep knowledge of the microarchitecture, often not available, and involves identifying shared resources and reverse-engineering their functions [4]. This process is time-consuming and ineffective for unknown resources.

SpecScope, our introduced framework, automates the discovery of contention-based Spectre gadgets. These gadgets exploit stateless resources in speculative execution. SpecScope's main aim is to minimize false positives by focusing on the contention between transient and non-transient instructions. It consists of two phases: generating contention maps showing interaction levels between instruction pairs, and using these maps in a static code analysis tool to find Spectre gadgets in code. This approach bypasses the need for complex reverse-engineering. Unlike SMoTherSpectre [1], SpecScope doesn't limit itself to port contention and, unlike SMoTherSpectre and ABSynthe [4], it reduces false positives by using transient instruction contention.

We can summarize our contributions to this work as follows:

- We present a framework for generating complete contention maps between transient and non-transient executions given a black-box microarchitecture.
- We describe a methodology and developed a static analysis tool that uses the contention maps to find exploitable contention-based Spectre gadgets in a given program.
- Our analysis shows that our framework can find contentions beyond execution ports (provides more coverage compared to SMoTherSpectre) as well as filtering contentions that cannot be used in Spectre attacks (avoids false positive gadgets).
- Our results show patching programs against Spectre can be microarchitecture-dependent to reduce the defense overhead while providing the same security level.
- Last but not least, our tool is open-sourced and publicly available to the community.

II. Related Work

The research community has put forth various efforts to enhance the automation of Spectre gadget discovery. We have

979-8-3503-4859-0/DATE24/© 2024 EDAA

summarized the recent works in Table I and compared them from different points of view. Efforts have been made by SpecTaint [10] and SpecFuzz [11] to automate the discovery of Spectre gadgets through dynamic analysis techniques. However, their capabilities are confined to identifying Spectre variants reliant on stateful microarchitectural resources for covert channel execution. On a different front, SMoTherSpectre [1] introduced an automated tool designed to uncover Spectre gadgets exploiting contention within the execution port. Nevertheless, this tool operates under the assumption that instructions using the same port will consistently exhibit contention, which is not always the case. Furthermore, it may overlook gadgets that do not rely on port contention. Building upon these advancements, ABSynthe [4] has expanded the capabilities of SMoTherSpectre to automatically identify instructions contending for the same execution port.

Osiris [12] is a microarchitectural side channel discovery tool that does not specifically target transmit gadgets and tends to have a higher false-negative rate when identifying Spectre gadgets. In contrast, Kasper [13] utilizes taint analysis policies to simulate an attacker's behavior on a transient path in the Linux kernel and system calls, not the user program. Moreover, its port contention vulnerability scanner lacks empirical knowledge, resulting in higher True-Negative gadgets. SPECTECTOR [14] automatically detects speculative non-interference through symbolic execution. SPECTECTOR is constrained by the limitations of symbolic execution and may sacrifice the soundness and completeness of analysis when dealing with large programs.

TABLE I: Comparison of state-of-the-art works

Technique	Analysis	Spectre Gadget Finder	Exploitable Resources	Platform Knowledge
SMoTherSpectre	Static	Yes	State-less (execution port)	White Box
ABSynthe	Dynamic	No	State-less (contention)	Black Box
Osiris	Dynamic	Yes	Stateful/less	White Box
Kasper	Dynamic	Yes	Stateful/less	White Box
SpecScope	Dynamic + static	Yes	State-less (on/cross-core)	Black Box

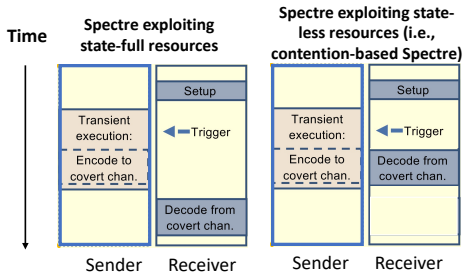

Fig. 1: Illustrating how the covert channels are used in stateless and stateful resources and how they can be running transiently

III. SpecScope

A. Overview

Transient execution attacks are possible when covert channels are combined with transient execution. In such attacks, the secret data is only available for the duration of the speculation window. During transient execution, the secret data can be accessed and encoded into a covert channel, which can then be decoded by the attacker later. We can divide the transient execution attacks into three phases [15]:

Setup Phase: This phase consists of two parts. First, the attacker prepares the microarchitecture to allow entering transient execution so that secret data can be accessed. Second, the attacker should prepare the microarchitectural transmission channel for recovering the data later.

Transient Execution Phase: Transient execution of the leak gadget occurs in this phase due to the training received during the setup phase. First, the attacker should trigger transient execution. Then, the transient instructions will be executed either in the victim or the attacker domain and the attacker prepares the data for transmission.

Decoding Phase: In this phase, the attacker can extract the encoded data. Traditionally, the data would be encoded in the cache, and the attacker can deploy cache-based side channels for retrieving data. However, the attacker can retrieve the data by activating a transmitting gadget and observing the behavior.

A novel aspect of our work involves exploring the capability of a transient transmit gadget to function as a covert channel, an idea that has not been investigated in previous works. Figure 1 illustrates the novelty of our gadget finder when the covert channel is transiently running and it uses a stateless resource to transmit the secret data. After the end of the speculation window, there is no way for the receiver in the attacker app to communicate with its sender. On the other hand, the other approaches are not able to detect such gadgets as they are not aware of the transient covert channel and do not generate a transient contention map. This distinguishs our work from the rest of the techniques that are finding Spectre gadgets.

To achieve this goal, SpecScope has two main phases, as shown in Figure 2. *Phase 1*, contention maps generation, automatically finds all possible contention-based channels that can be used in a Spectre attack for a given black-box target microarchitecture. This phase depends on observing the contention between transiently executed instructions and non-transiently executed instructions.*Phase 2*, contention-based Spectre gadget finding, is a static code analysis tool that depends on the contention maps to find contention-based Spectre gadgets that are specific to the target microarchitecture in a given code, e.g., public library. More importantly, SpecScope does not require the knowledge of the target microarchitecture resources to exploit the microarchitectural vulnerabilities.

B. Threat Model

In this work, we assume that the targeted platform has hyperthreading capability and it is enabled. The microarchitecture is vulnerable to known (such as Spectre and Meltdown) or even unknown (futuristic – not yet discovered) transient execution attacks. The user has black-box access to the targeted microarchitecture while can run programs on the microarchitecture and the ISA instructions are known to the user. In

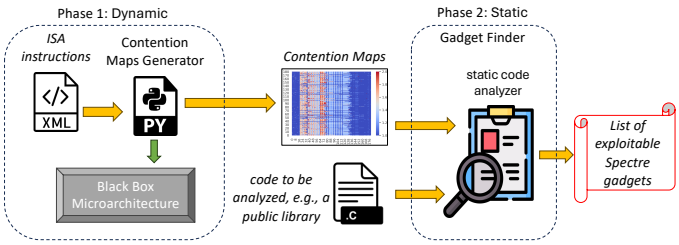

Fig. 2: Block diagram of SpecScope

addition, the user is not required to perform the measurements in isolation (without having other processes running on the system, for example, if the targeted microarchitecture is in a shared cloud environment) to create the contention map; the impact of other factors would be diminished when the measurements are repeated hundreds of times and the averaging would remove the outliers' impact.

C. Contention Maps Generation

To generate contention maps for a microarchitecture, we analyze the execution time of parallel instructions to detect resource contention. When two instructions, A and B, execute on a hyperthreading-enabled core, their execution time reflects their shared resource usage. If A and B use the same microarchitectural resource, like a functional unit, their combined execution time increases. This method identifies execution time interference between parallel instructions.

SpecScope, unlike ABSynthe, focuses on interference between transiently executed and non-transiently executed instructions. In transient execution attacks, such as Spectre, sensitive data is accessed during transient execution and transmitted through a covert channel, with the receiving end being a non-transient instruction. Therefore, SpecScope examines all possible interference scenarios within a microarchitecture's ISA, considering transient execution of one instruction and non-transient execution of others.

Our framework measures contention by running a transient execution instruction in a 'sender' thread and a non-transient instruction in a 'receiver' thread, observing potential contention. We measure the receiver thread's execution time for a test instruction, both with and without the sender's concurrent execution. For example, to assess interference between two instructions S (transient) and R (non-transient), we compare their execution times when run together versus separately.

$$Interference = \frac{Exe.\,time\,of\,R\,while\,S\,running\,simultaneously}{Execution\,time\,of\,R}$$

If $Interference$ is bigger than a $threshold$ then contention is observed, and the instructions can be used in transient execution attacks. In section IV-B, we explain how we calculate the $threshold$. The interference threshold is the minimum contention value that can be used to construct a reliable covert channel, i.e., selects which contentions should be used to find gadgets in the second phase of our proposed framework.

The sender process includes an $lfence$ instruction after poisoning. $lfence$ Performs a serializing operation on all load-from-memory instructions issued before the $lfence$ instruction that helps to reduce the interference noise from the branch poisoning instructions. Subsequently, the sender signals the receiver process (Synchronization/Handshake phase) to start execution (using shared memory), and triggers the transient execution. For completeness, we describe the steps taken in each running thread (sender and receiver) for a single iteration separability:

On the sender thread: for each instruction k in the ISA, sender perform the following sequence N_i times:

- First, the sender poisons the Branch History Table.
- Then, the sender sends a ReadyToSendSig by writing 1 to shared memory.
- Next, it waits on the ReadyToReceiveSig to ensure the receiver is also ready.
- Subsequently, it triggers transient execution using malicious input to the conditional jump and poisoned BHT to speculatively execute the fall through instructions if the conditional jump.
- Finally, execute instruction k multiple times (N).

Correspondingly, on the receiver thread: for each instruction j in the ISA:

- The receiver waits for the sender to get ready (polling shared memory).
- When it ensures that the receiver is ready, then it sends the ReadyToReceive signal. In this way, synchronization is done between the sender and receiver.
- Receiver reads the current time using *rdtscp* instruction.
- Receiver executes instruction j multiple times (N).
- Eventually, the receiver reads the time to calculate the delay and writes it to a file.

The final product of this step is a contention map for the given micro-architecture that we ran our framework on. A contention map is a 2D array that each row and column of the array is an instruction from ISA. Each element of the array e_{ij} is $Interference$ value, described above, that shows the amount of delay imposed on the instruction i while instruction j was simultaneously executed.

D. Gadget Finder

To identify contention-based Spectre gadgets, our methodology utilizes contention maps, which indicate the level of interference between each transiently executed instruction and all non-transiently executed instructions. By statically analyzing code and examining each conditional branch, we determine if it can potentially leak its input data. A gadget is deemed vulnerable if it meets two conditions: (1) it shows no contention with instructions on the alternate path of the branch, and (2) it exhibits contention with any ISA instruction. The first condition ensures the gadget can leak data from the branch since it's unique to one path and uncontentious with the other. The second condition identifies instructions an attacker could use to leak data. These conditions are verified by examining the contention map entries for the tested transient instruction against instructions in both branch paths.

Implementation details: *Distorm3* disassembler has been used to analyze the target program for finding gadgets.

For finding a contention-based Spectre gadget, our tool has two inputs, which are: (i) *code*: the code that tool will analyze, e.g., a public library, and (ii) *contention_map*: contention map generated for a specific micro-architecture. The tool will first identify all conditional branches in the given code (*code*). Then for each conditional branch in the code, the tool will take two main steps:

- (Step 1) Branch blocks identification: in this step, the tool identifies the fall-through path and the branch target path.
- (Step 2) Contention analysis: in this step, the tool identifies if the branch is a contention-based Spectre gadget or not, based on the instructions in the branch fall-through and target paths as well as the contention map. Specifically, the contention map will be used to analyze the instructions in both paths of the branch. In particular, for each instruction, we check the aforementioned two conditions. If both conditions are satisfied, then we consider the branch as a vulnerable gadget. However, if these two conditions are not satisfied by any instruction in both paths of the branch, then we consider the branch as non-vulnerable.

It should be mentioned that both SMoTherSpectre and SpecScope use a static analysis tool to find conditional jumps and decide whether this should be labeled as a Spectre gadget. The difference is in how they decide. SMoTherSpectre uses Intel Architecture Code Analyzer to determine the execution port for each instruction and assumes if two instructions use the same port then they can be used to create contention. However, SpecScope uses the contention maps to determine the instruction that can be used to create contention. This resulted in two main differences: 1) SMoTherSpectre finds gadgets that use ports contention as a covert channel, while our work generalizes to find gadgets that use ports or cross-port contentions. 2) SMoTherSpectre, has more false positives since they assume if two instructions use the same port then they have a contention. However, not all instructions can run speculatively which leads to finding false positive gadgets since gadgets based on these instructions cannot be used to construct a Spectre attack.

IV. Evaluation

A. Experimental Setup

We run our experiments on three different Intel architectures, which are SkyLake, KabyLake, and Nehalem. All CPUs are running Ubuntu 18.04 LTS, kernel version 4.15.0. To reduce the setup noise, the CPU frequency is set to a maximum clock. The power state of the processor and NUMA feature are always disabled. Apart from these changes, all other settings are kept to their defaults.

B. Characterization of Contention Maps Generation

SpecScope primarily focuses on determining whether two instructions, one executed transiently while the other is executed non-transiently, can have observable contention when they are executed simultaneously. To accomplish this, we evaluated the methodology discussed in Section III-C. Figure 3-(a) shows the contention maps for both Nehalem and Skylake CPU architectures. In particular, this figure shows that there is observable contention between transiently executed instructions and non-transiently executed instructions. We generated contention maps for the same CPU architectures when both sender and receiver execute instructions non-transiently. The results are shown in 3-(b) and demonstrate that there are more observable contentions when the sender uses non-transient instructions. Our analysis shows that the reason is that not all instructions can execute transiently, e.g., store instructions only execute at the commit stage. In addition, this result verifies that the contention maps generated using SpecScope only show the channels that can be used by Spectre attacks rather than all possible covert channels on that microarchitecture.

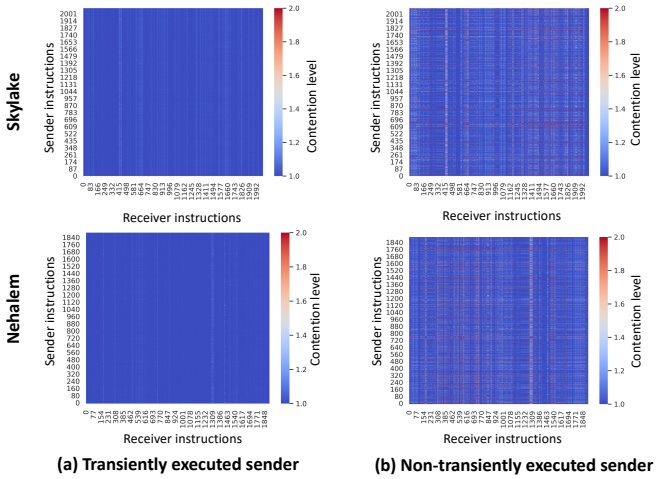

Fig. 3: Contention of ISA instructions on the same core

C. Quality of Contention

We conducted experiments to assess the interference observed within our contention maps. To ensure a fair comparison of all instructions, we computed the execution times of receiver (*R*) instructions when they ran either with a *NOP* or concurrently with the sender (*S*) transiently executed instruction. These calculated times were then normalized to fall within the range of 0 to 1 for each instruction. Our findings revealed that instructions running without contention exhibited lower standard deviation in their execution times compared to those experiencing contention. For instance, on a Skylake CPU running at its highest frequency, instructions with contention had an average standard deviation almost 2.8 times higher.

Based on this observation, we can identify whether two instructions (*S* and *R*) contend solely by examining the standard deviation. This approach enhances the accuracy of the contention channel without the need to consider the absolute value of the instruction's execution time. In comparison to threshold-based methods [4], [1], standard deviation-based detection yields very low False Positive and False Negative rates, standing at 1.66% and 0.11%, respectively. However, in scenarios where conducting multiple experiment runs is not feasible, the threshold-based approach remains valuable.

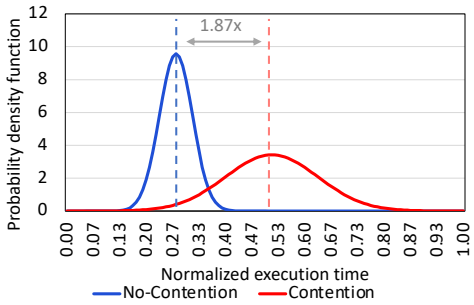

Fig. 4: Normalized exe. time of instructions while running w/wo contention

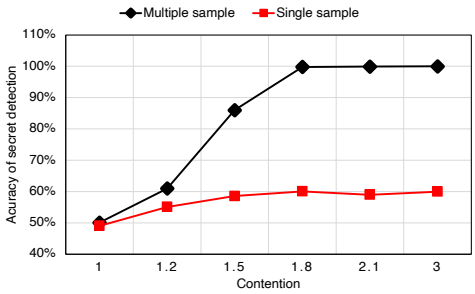

Fig. 5: Acc. of secret bit detection based on the contention.

One noteworthy finding is that, on a Skylake CPU running at its highest frequency, instructions ran on average 1.87 times faster when not contending. The results of our analysis are depicted in Figure 4. It's important to note that values such as 1.87X are specific to the Skylake CPU and should be determined through the same experiment for other CPUs, making it a one-time task per microarchitecture.

To assess the threshold, we conducted another experiment, where we assigned a bit value of 1 to contention and a bit value of 0 to no contention. Subsequently, we transmitted 10,000 randomly generated bits and decoded them at the receiver side. This experiment was conducted in two scenarios: 1) In the first scenario, we transferred only one sample for each bit. 2) In the second scenario, we transferred multiple samples (specifically, 5 in our experiment) for each bit. Figure 5 illustrates the accuracy of bit detection when mapped to contention. We observed that when the threshold exceeds 1.8 (as observed with Skylake running at its maximum frequency), the accuracy for multiple samples approaches the ideal 99.97%.

We conducted an analysis of the impact of CPU frequency on the interference threshold needed to attain 95% accuracy in secret bit detection, and the findings are depicted in Figure 6. This figure illustrates that when the CPU operates at a higher frequency, a correspondingly higher interference threshold is necessary to achieve the same attack success rate as when the CPU is running at a lower frequency, as a sign of a clear correlation between CPU frequency and the required threshold. Additionally, we observed that the SkyLake-AVX512 microarchitecture demands a higher threshold compared to KabyLake and Nehalem since it has a larger number of cores and computational units, which can introduce more noise into the system.

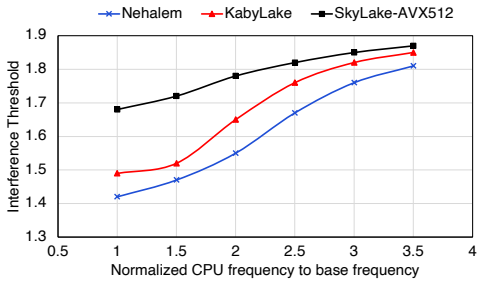

Fig. 6: Relation of inter. threshold with the freq.

D. Proof of Concept

In this section, we evaluate the performance of SpecScope in finding exploitable gadgets and compare it with the recent tools. We analyzed public libraries existing in regular Ubuntu installation. Table II illustrates the number of gadgets found by SpecScope on Intel Nehalem, KabyLake, and Skylake-AVX512 microarchitectures and compares them with state-of-the-art. The libraries that we used are based on what has been used in recent related works. SpecFuzz and SpecTaint are not focusing on finding contention-based gadgets, thus listing the number of gadgets using them is not relevant. As they find orthogonal gadgets, they could be used along with our tool to be able to find all possible gadgets.

In comparison with other tools, our results show additional gadgets due to existing contention in other shared resources rather than ports and our tool can find even transient transmit gadgets. For example, in our Skylake experimental platform, out of 8,673,025 instruction couples, only 2.8% of them had observable contention, and by simply investigating the utilization of these instruction pairs within the public library, we uncovered several previously unrecognized potential gadgets. Our results highlight that a single program may exhibit different vulnerable gadgets on each microarchitecture. Notably, SpecScope identified varying numbers of gadgets for each library on the Kabylake and Skylake architectures.

It's worth noting that not all of these gadgets are capable of leaking sensitive data. Therefore, further analysis is required to determine which of them constitute vulnerable gadgets. To rank the gadgets detected by SpecScope for assigning a priority for the patching, we considered two parameters: the length of the gadget and contention intensity. We can divide both the length and intensity into two groups (short/long) and (low/high). We consider the intensity high if the contention delay is 1.8× more than normal execution. The length also is considered long if it is longer than 80% of the speculative execution's window. It is evident that the definitions provided above can vary depending on the specific level of security requirements.

According to this classification, we group the gadgets into four categories: critical, important, trivial, and incidental. A gadget falls into the critical category if it has a lengthy code sequence and exhibits high contention intensity between its target and fall-through paths. Gadgets are categorized as important when they demonstrate high contention intensity but have a shorter length. Similarly, gadgets are classified as trivial

TABLE II: Number of gadgets in popular public libraries.

Library	Total jumps	SpecScope (Kabylake)	SpecScope (Skylake)	SpecScope (Nehalem)	SMoTherSpectre
glibc 2.23	65039	28581	25390	19701	15382
stdc++ 6.0	36668	9809	10503	10675	3747
ld 2.23	6523	2120	1455	1996	1316
ssl 1.1	11086	3377	3209	3499	1826
crypto 1.1	67689	22802	19231	19343	8774

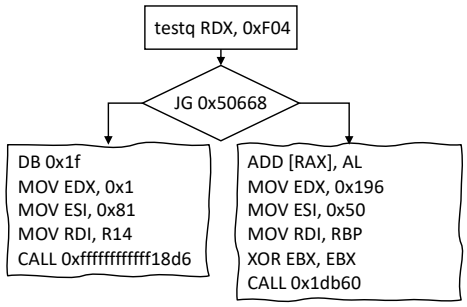

Fig. 7: A new gadget discovered only by SpecScope on Nehalem platform in ssl 1.1 library.

if their contention intensity is low, but they possess a lengthy code sequence. Lastly, incidental gadgets are characterized by both low intensity and a short code length. Following our defined criteria, it was found that only 6% of the gadgets identified by SpecScope could be considered critical, on average. Approximately 23% of the gadgets fall into the important category, while 34% are categorized as trivial. The remaining gadgets belong to the incidental category.

For a fair comparison, we employed SMoTherSpecter's proof of concept to attack its reference implementation of the test program that utilizes OpenSSL for data encryption. To leverage our discovered gadgets (identified by SpecScope and categorized as critical), we linked both the attack and victim to the *libcrypto* library. Subsequently, we injected our selected gadgets (one presented in Figure 7) into the victim, and the attacker's timing sequence was injected into the same memory address within its binary. It's important to clarify that introducing the gadget into the victim's binary primarily served the purpose of establishing its static address. In our setup, synchronization between the attacker and victim occurred solely prior to the encryption function call, with the attacker's timing sequence commencing after a specified delay. With an equal number of encryption runs (100K), the attacker achieved a success rate of up to 98% in detecting the victim's secret bit using our gadgets, compared to SMoTherSpectre's success rate of 80% on the Nehalem platform.

We also conducted a comparison of the gadgets identified by SpecScope on Skylake with those found by SMoTherSpectre, using confusion metrics, presented in Table III.True Positive (TP) column indicates the rate of gadgets found by the other tool that match with those found by SpecScope. False Negative (FN) represents the rate of SpecScope's gadgets that do not match with gadgets found by the other tool. False Positive (FP) reflects the rate of the other tool's gadgets that do not match with those identified by SpecScope.

TABLE III: Rates of TP, FP, and FN in comparison to SpecScope on the Skylake microarchitecture.

Library	**SMoTherSpectre**		
	TP	FP	FN
glibc 2.23	0.51	0.11	0.38
stdc++ 6.0	0.29	0.13	0.58
ld 2.23	0.77	0.09	0.14
ssl 1.1	0.48	0.13	0.39
crypto 1.1	0.41	0.14	0.45

V. Conclusion

Spectre attacks are an enormous security threat since they can read arbitrary data from other security domains. This work presents SpecScope, a system designed for automatic identification of Spectre gadgets in software running on a black-box microarchitecture. SpecScope focuses on the interaction between transient and non-transient instructions to accurately find Spectre gadgets and reduce the false-positives. Through an experimental evaluation, we demonstrated that SpecScope can automatically discover practical Spectre gadgets in a variety of public libraries.

VI. Acknowledgment

The work in this paper is partially supported by National Science Foundation grants CNS-2155002 and CNS-2155029.

References

[1] A. Bhattacharyya *et al.*, "Smotherspectre: exploiting speculative execution through port contention," in *ACM SIGSAC CCS*, 2019.
[2] N. Nazari *et al.*, "Adversarial attacks against machine learning-based resource provisioning systems," *IEEE Micro*, 2023.
[3] H. M. Makrani *et al.*, "Cloak & co-locate: Adversarial railroading of resource sharing-based attacks on the cloud," in *SEED*, 2021.
[4] B. Gras *et al.*, "Absynthe: Automatic blackbox side-channel synthesis on commodity microarchitectures." in *NDSS*, 2020.
[5] K. N. Khasawneh *et al.*, "Safespec: Banishing the spectre of a meltdown with leakage-free speculation," in *ACM/IEEE DAC*, 2019.
[6] S. Ainsworth *et al.*, "Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state," in *ACM/IEEE ISCA*, 2020.
[7] M. S. Islam *et al.*, "Nd-hmds: Non-differentiable hardware malware detectors against evasive transient execution attacks," in *ICCD*, 2020.
[8] A. Pardoe, "Spectre mitigations in msvc," 2018.
[9] C. Liu *et al.*, "Frequency throttling side-channel attack," in *CCS*, 2022.
[10] Z. Qi *et al.*, "Spectaint: Speculative taint analysis for discovering spectre gadgets," 2021.
[11] O. Oleksenko *et al.*, "Specfuzz: Bringing spectre-type vulnerabilities to the surface," in *USENIX Security*, 2020.
[12] D. Weber *et al.*, "Osiris: Automated discovery of microarchitectural side channels," in *USENIX Security*, 2021.
[13] B. Johannesmeyer *et al.*, "Kasper: scanning for generalized transient execution gadgets in the linux kernel," in *NDSS*, 2022.
[14] M. Guarnieri *et al.*, "Spectector: Principled detection of speculative information flows," in *IEEE Security and Privacy*, 2020.
[15] C. Canella *et al.*, "The evolution of transient-execution attacks," in *Proceedings GLSVLSI*, 2020.