GadgetMeter: Quantitatively and Accurately Gauging the Exploitability of Speculative Gadgets

Qi Ling*¶, Yujun Liang†, Yi Ren‡, Baris Kasikci§‖ and Shuwen Deng†✉
*Department of Computer Science, Purdue University
‡Institute for Interdisciplinary Information Sciences, Tsinghua University
§Paul G. Allen School of Computer Science & Engineering, University of Washington, ‖Google
†Department of Electronic Engineering, Tsinghua University
ling102@purdue.edu, {liang-yj21, yi-ren20}@mails.tsinghua.edu.cn, baris@cs.washington.edu, shuwend@tsinghua.edu.cn

***Abstract*—Since their emergence in 2018, speculative execution attacks have proven difficult to fully prevent without substantial performance overhead. This is because most mitigations hurt modern processors' speculative nature, which is essential to many optimization techniques. To address this, numerous scanners have been developed to identify vulnerable code snippets (speculative gadgets) within software applications, allowing mitigations to be applied selectively and thereby minimizing performance degradation.**

In this paper, we show that existing speculative gadget scanners lack accuracy, often misclassifying gadgets due to limited modeling of timing properties. Instead, we identify another fundamental condition intrinsic to all speculative attacks—the timing requirement as a race condition inside the gadget. Specifically, the attacker must optimize the race condition between speculated authorization and secret leakage to successfully exploit the gadget. Therefore, we introduce GadgetMeter, a framework designed to quantitatively gauge the exploitability of speculative gadgets based on their timing property. We systematically explore the attacker's power to optimize the race condition inside gadgets (windowing power). A Directed Acyclic Instruction Graph is used to model timing conditions, and static analysis and runtime testing are combined to optimize attack patterns and quantify gadget vulnerability. We use GadgetMeter to evaluate gadgets in a wide range of software, including six real-world applications and the Linux kernel. Our result shows that GadgetMeter can accurately identify exploitable speculative gadgets and quantify their vulnerability level, identifying 471 gadgets reported by state-of-the-art works as unexploitable.

I. Introduction

Speculative execution attacks have emerged as a prevailing security concern since 2018 [9], [13], [17], [18], [25], [32], [35], [37], [39], [43], [46], [52], [55], [56], [60], [61], and their mitigation poses an intricate challenge for modern processors. These attacks take advantage of the speculative nature of modern processors, e.g., conditional and indirect branch predictions, which is a fundamental principle to many optimization techniques. This widespread adoption implies that a wide spectrum of processors across multiple manufacturers are at potential risk [35]. The menace of speculative attacks extends across platforms, threatening various devices from servers and mobile phones to embedded systems [12]. The situation is further complicated by the frequent emergence of new variants, empowering malicious entities to extract sensitive information like passwords directly from the hardware [26]. Furthermore, efficacious mitigation solutions often compromise CPU performance, introducing significant practical implementation overhead [37], [60], e.g., as of September 2022, ESXi VM[1] performance has seen a decline of up to 70% due to Intel's Retbleed mitigation measures [2], [60].

As described in studies [14], [23], [51], [66], speculative execution attacks have three key phases. First, during the *initial setup phase*, the attacker configures architectural and microarchitectural states to ensure the victim program executes transiently in a controlled manner. For instance, in a Spectre-V1 attack [35], the attacker trains the branch predictor to mispredict the target branch and flushes the victim's boundary value from the cache to expand the speculation window. Second, during the *transient execution phase*, the victim code executes transiently, and while architectural state changes are rolled back at misspeculation, microarchitectural state changes remain. Therefore, any secret-dependent traces can act as covert channels to leak data. Finally, in the *decoding phase*, the attacker retrieves the secret data from these covert channels.

Speculative Gadget. While the setup and decoding phases can be achieved with the execution of attacker code, the transient execution phase must be performed by some victim code, which we call a *speculative gadget*. For a victim code snippet (gadget) to be qualified as a speculative gadget, it must satisfy several requirements so that the attacker can mount a speculative execution attack on it. We summarize these requirements into two root conditions of speculative gadgets as described below:

- Information flow condition: On the misspeculation of a set of instructions, there exists a propagation datapath on the transient execution path which empowers the attacker to control the victim's execution to access a secret and leak it transiently through a covert channel. We refer to

✉Shuwen Deng is the corresponding author.
¶Qi Ling is affiliated with Purdue University, but was at Shanghai Jiao Tong University during this work.

[1]VMware ESXi is an enterprise-class, type-1 hypervisor developed by VMware for deploying and serving virtual computers [30].

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.241723
www.ndss-symposium.org

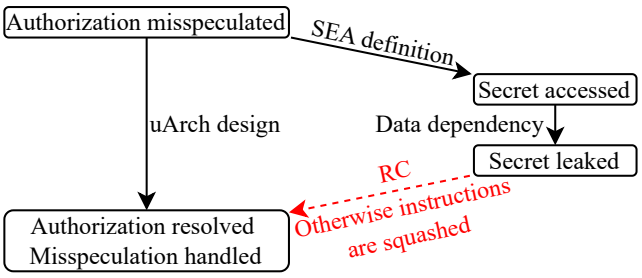


Fig. 1: Race condition between the resolution of authorization instructions and leakage instructions in speculative execution attacks (SEA). Each edge represents a happens-before relationship between two attack events, and the label is the cause. For a gadget to be exploitable, the attacker must optimize the race condition to ensure secret data is transmitted before the authorizations are resolved and misspeculations are handled.

```
1 x = user_input(); {   // Injects values
2 if (x < 16) {         // Misp. branch
3   y=array1[x];        // Accesses secret
4   z=array2[512*y];}   // Leaks secret
```

Listing 1: Gadget misclassified as vulnerable by SpecFuzz [48], Spectaint [50], and Kasper [33]. Different from classical Spectre-V1 proof-of-concept code, with the condition bound to be a constant number, the conditional branch at line 2 will be resolved shortly after `x` becomes available, while the disclosure gadget at lines 3 and 4 is data dependent on `x` and takes longer to execute. Therefore, the speculative execution will be terminated well before the disclosure gadget leaks the secret.

the instructions that are misspeculated inadvertently as the *authorization instructions*. We refer to the instructions that leak secret transiently as the *leakage instructions*. For example, in Proof-of-Concept (PoC) Spectre-V1, a speculative gadget must contain a branch instruction where the misprediction will lead to an attacker-controlled memory access to secret data.

- Timing condition: The gadget should have instructions arranged in a way that the attacker can optimize the race condition between authorization and leakage instructions. As shown in Figure 1, to successfully exploit a gadget, the attacker must force the leakage instructions to execute before the resolution of misspeculated authorizations. Otherwise, the leakage instructions will be squashed before leaking any secrets, making the attack fail.

Gadget Detection Challenges. Extensive research [28], [33], [42], [48], [50], [57] has focused on scanning for speculative gadgets within programs to mitigate speculative execution attacks. Unfortunately, existing gadget scanners have limited soundness as they often report gadgets that are hard to exploit. There are mainly two reasons.

First, they limit their focus on the information flow property of gadgets but ignore the timing property. Existing scanners apply a wide range of techniques to achieve accurate and scalable detection of vulnerable transient information flow within a victim application [16]. However, most of them do not accurately model the timing property of gadgets. Instead, they consider a gadget to satisfy the timing condition as long as the misspeculated authorization instructions and the transient leakage instructions can fit into the Re-Order Buffer (ROB). This modeling, as demonstrated in Listing 1, *over-approximates the exploitability of gadgets, leading to false positives*.

Second, some gadget scanners try to model the timing condition but are limited in scope. For example, SpecCheck [42] is capable of modeling the length of the speculation window in the cycle-accurate processor simulator Gem5, but it fails to model an attacker optimizing the race condition. Some other works [20], [29], [63] consider a memory-dependent branch as necessary for a gadget to be exploitable, which is untrue as demonstrated in Listing 2d. The reason behind these design failures lies in a lack of study on the *windowing power*, which we defined as the attacker's power to optimize the race condition in gadgets. So far, most attacks simply force a cache miss on the direct data dependency of authorization instructions, such as the boundary value in Spectre-PHT and the jump target address in Spectre-BTB. There is a limited understanding of what makes a gadget exploitable on the scanner side.

GadgetMeter Design. In this paper, we propose GadgetMeter, a novel framework to quantitatively evaluate the exploitability of speculative gadgets based on their timing property. Unlike existing scanners, GadgetMeter emphasizes assessing and prioritizing gadget exploitability rather than merely locating gadgets. This approach enhances the effectiveness of state-of-the-art scanning tools by providing a deeper analysis of gadgets' practical exploitability.

To accomplish this, we first present a systematic exploration of the attacker's *windowing power*. We view the windowing power as an interplay between the windowing capability and the windowing strategy. For capability, the attacker can apply a wide range of techniques beyond cache line eviction to optimize the race condition. We systematically review potential capabilities, drawing insights from microarchitectural timing attacks, and focus on the most powerful ones. For strategy, the attacker can employ a sophisticated attack pattern, rather than naively targeting all data dependencies or brute force to get the best pattern. By exploring both aspects, we discover a large space of previously overlooked windowing power. We then evaluate these windowing powers based on a series of novel metrics first described in this work.

Leveraging these insights, we design GadgetMeter as a combination of static analysis and runtime testing.

- In Step A, to *model the timing condition* of a gadget, we introduce a ***Directed Acyclic Instruction Graph*** to represent the gadget pattern and describe the timing condition. We perform two depth-first searches from the misspeculated branch instruction and the leaking instruc-

tions respectively, which are then merged into one DAG to describe the timing condition.
- In Step B, GadgetMeter *simulates the attacker* by constructing a ***windowing-power-optimized analytical model*** and performing static analysis to select an effective attack pattern for each target gadget.
- In Step C, to ***quantify the exploitability*** of the gadget, GadgetMeter performs *runtime testing* to derive a vulnerability score, accurately representing the practical exploitability.

We evaluate GadgetMeter on a variety of benchmarks, encompassing Kocher's micro-benchmark [5], 6 security-centric applications, and the Linux kernel. GadgetMeter effectively identifies 471 unexploitable gadgets compared with state-of-the-art scanners [33], [48]. Also, GadgetMeter can quantify the vulnerability level of different gadgets while existing scanners consider them as equally vulnerable. We provide three case studies to demonstrate the capability of GadgetMeter.

To summarize, we make the following contributions:
- Provide a systematic exploration and evaluation of windowing power for speculative execution attacks.
- Model the timing condition of speculative gadgets with a Directed Acyclic Instruction Graph, where to the best of the author's knowledge, similar methods are not utilized for state-of-the-art gadget scanners.
- Propose GadgetMeter, a comprehensive and quantitative analysis framework for evaluating and prioritizing gadgets' exploitability.
- Evaluate on security-centric applications, identifying 471 gadgets reported by state-of-the-art works as unexploitable.

The code used in this paper will be released under open-source license at https://github.com/qiling07/GadgetMeter.git.

II. Background and Related Work

In this section, we will first provide background on the components involved in speculative execution attacks, motivating the need for a systematic exploration of windowing power. Next, we will provide background on existing speculative gadget scanners, highlighting their limitations and the need for a gadget scanner that is aware of the timing condition.

A. Speculative Execution Attacks

To expand the attack surface, numerous studies have explored new ways to perform each step in a speculative execution attack.

Speculation mechanism. Existing attacks leverage various speculative mechanisms in modern processors. Spectre-V1 [35], NetSpectre [52], and SpectreRewind [25] exploit the pattern history table (PHT) for conditional branch prediction, making them hard to mitigate without performance impact [33]. Other microarchitectural features have been exploited as well, such as branch target buffer (BTB) [13], [17], [35], store-to-load (STL) dependency prediction [1], return stack buffer (RSB) [37], [39], branch type prediction [61], and predictions in string and division instructions [46].

TABLE I: Comparison of our work and prior works. Works marked by * only simulate a naive windowing power, where the attacker blindly evicts all memory dependencies of branch instructions out of cache.

Scanners	Model Timing Condition	Model Windowing Power	Quantify Exploitability
Spectector [28]	✗	✗	✗
oo7 [57]	✗	✗	✗
SpecFuzz [48]	✗	✗	✗
SpecTaint [50]	✗	✗	✗
Kasper [33]	✗	✗	✗
SpecCheck [42]	✓	✗	✗
Haunted [20]	✓	✓*	✗
Wu [63]	✓	✓*	✗
SpecuSym [29]	✓	✓*	✗
Our work	✓	✓	✓

Secret leakage channel. Existing attacks use various side/covert channels for transient secret leakage. Cache-based side channels are common [6], [35], [37], [60], while other attacks leverage execution port [13], FP division unit contention [25], AVX2 timing [52], microarchitectural data sampling [15], and physical channels [18], [43].

Windowing power. Only a few works have explored advanced windowing power to optimize the timing condition (Figure 1) in speculative execution attacks. So far, most attacks employ cache set eviction to force a cache miss on the direct data dependencies of authorization instructions. Examples include cache eviction of boundary value in Spectre-PHT [17], [35], [43], [55], jump target address in Spectre-BTB [13], [35], [41], store target address in Spectre-STL [1], and return address in Spectre-RSB [37], [39]. BranchSpec [32] explores more methods to delay victim memory accesses, including TLB misses, longer page-walk paths, and page faults. Moreover, Milburn et al. [44] first uses SMT execution port contention to congest the indirect branch in Spectre-BTB, proving the `lfence/jmp` mitigation ineffective.

B. Speculative Gadget Scanners

To efficiently mitigate speculative execution attacks, extensive research has been directed toward scanning for speculative gadgets within programs, especially for Spectre-PHT. The focus on Spectre-PHT arises because other variants such as Spectre-RSB [37], [39], Spectre-BTB [35], and Spectre-STL [1], can typically be mitigated with minimal overhead using microcode updates [19] or software updates [34], [70]. However, mitigating Spectre-PHT poses a more significant challenge. While adding an *lfence* instruction after every conditional branch can prevent Spectre-PHT attacks, this approach introduces up to 440% overhead on Phoenix benchmarks [47].

To minimize the mitigation overhead, software developers can employ speculative gadget scanners and apply patches only to exploitable gadgets. Spectector [28] and oo7 [57] perform static taint analysis and symbolic execution to search for vulnerable information flow. SpecFuzz [48], SpecTaint [50], and Kasper [33] all perform dynamic analyses like speculation

exposure and dynamic taint analysis to improve the accuracy and scalability.

Unfortunately, existing gadget scanners lack soundness and often report hard-to-exploit gadgets. They focus on information flow properties but fail to accurately model the timing properties of gadgets. Most scanners consider a gadget valid as long as the misspeculated authorization instruction and the leakage instruction fit into the Re-Order Buffer (ROB), leading to false positives as shown in Listing 1.

Some gadget scanners attempt to model the timing condition but are limited in scope. For example, SpecCheck [42] is capable of modeling the length of the speculation window in the cycle-accurate processor simulator Gem5, but it fails to model an attacker optimizing the timing condition through actions such as cache line eviction. As a result, SpecCheck can fail to identify even the PoC Spectre gadget in Listing 2a if the boundary value happens to be in the cache. Some other works [20], [29], [63] consider a memory-dependent branch with cache miss as sufficient for a gadget to be exploitable. This view on timing condition is limited as a branch with cache miss does not necessarily make a gadget exploitable, as in Listings 2b and 2c. What is worse is that even a branch with no memory access can be exploitable, as seen in Listing 2d. We tested all gadgets from Listings 1 and 2 with various attack patterns. Listings 1, 2b, and 2c proved unexploitable with success rates under 0.51%, while Listings 2a and 2d were highly vulnerable with success rates over 99.8%.

Parallel to these scanners, Speculator [40] manually investigates the microarchitectural behavior of individual gadgets, while SpeechMiner [65] studies the microarchitectural timing of Meltdown vulnerabilities.

III. Threat Model and Scope

We focus on programs that, while deemed harmless, may be susceptible to speculative attacks. We consider a local, unprivileged attacker with the capability to invoke Application Programming Interface (API) calls for real-world applications and dispatch arbitrary system calls to a target kernel free of harmful software bugs. This attacker aims to harness a speculative gadget in these programs to leak their secret memory. In particular, the attacker is capable of performing speculative attacks utilizing a pattern history table, which triggers conditional branch misprediction and leaks secrets through side channels, including cache side channel, MDS, or port contention-based side channel. Meltdown-type attacks, which can be initiated from malicious programs without executing gadgets in victim software, fall outside the scope of our work.

IV. Systematic Study of Windowing Power

Windowing power, a key primitive in speculative execution attacks, is an interplay of two components: the windowing capability and the windowing strategy. The former dictates the operations the attacker can perform to influence the timing of the gadget execution, while the latter guides the attacker in selecting the most effective attack pattern to optimize the timing condition. A robust windowing capability equips the attacker with a set of unit operations capable of influencing the gadget execution by a significant number of cycles with fine-grained control. A well-crafted windowing strategy empowers the attacker to efficiently cherry-pick an optimal combination of unit operations, maximizing the timing condition when performed together.

In this section, we systematically study the attacker's windowing power by exploring the space of windowing capability and strategy, respectively.

```
1 // authorization has a unique mem access
2 x = user_input();
3 if (x < *boundaryPtr)
4   z = array2[512 * array1[x]];
```

(a) Gadget potentially missed by SpecCheck [42] if `*boundaryPtr` happens to reside in the cache during SpecCheck's gem5 simulation.

```
1 // authorization has a mem access
2 // which is also used by leakage
3 Object *obj = user_input();
4 if (obj->x < 16){
5   y = array1[obj->x];
6   z = array2[512 * y];}
```

(b) Gadget misclassified as exploitable by Haunted RelSE [20] and SpecuSym [29] because of the memory-dependent branch. However, forcing a cache miss on `&(obj->x)` will also delay the secret access on Line 5, leading to an unsuccessful attack.

```
1 // authorization has a mem access sharing
2 // a cache line with leakage's mem access
3 Object *obj = user_input();
4 if (x < obj->boundary){
5   y = obj->array[x];
6   z = array2[512 * y];}
```

(c) Gadget misclassified as exploitable by Haunted RelSE [20] and SpecuSym [29] because of the memory-dependent branch. However, forcing a cache miss on `&(obj->boundary)` will also delay the access to `&(obj->array)`, leading to a longer leakage time and unsuccessful attack.

```
1 // authorization has no mem access,
2 // but it depends on an FP division
3 x = user_input();
4 if (x < 32. / 2.)
5   z=array2[512 * array1[x]];
```

(d) Gadget exploitable in practice if the division unit is congested by the attacker, but misclassified by Haunted RelSE [20] and SpecuSym [29] for lack of a memory-dependent branch.

Listing 2: Gadgets that are misclassified by existing scanners for lack of analysis of their timing properties.

A. Windowing Capability

One fundamental reason for enabling windowing capabilities is a common hardware design choice: hardware resources are shared between different programs to improve parallelism and hardware utilization. However, this leads to a competition between programs sharing the same resource, empowering an attacker to slow down a victim by competing for the resource [54].

Many microarchitectural timing attacks [11], [38], [68] exploit the same hardware flaw by using techniques where one program's execution affects another's timing, which can be adapted for windowing capabilities. In these attacks, the sender encodes data by competing for shared resources, slowing down the receiver, who then decodes this information; this approach can be adapted for windowing by having the attacker as the sender to compete for shared resources and the victim as the receiver that is slowed down.

However, these techniques vary in effectiveness for windowing capabilities. For example, using memory bus and controller contention to exploit the PoC gadget in Listing 2a eventually delays both the boundary and secret accesses, thus failing the attack. Performing port contention in Listing 1 is also ineffective, as the latency delay is too minor to allow the transient execution of leakage instructions. In summary, a powerful windowing capability requires two key properties:

- Large latency effect: The windowing capability should delay the target operation by a large number of processor cycles.
- Fine-grained controllability: The attacker should be able to apply the windowing capability in a fine-grained manner. For capabilities affecting pipeline operations, the attacker can affect the latency of some types of instructions but not others. For capabilities affecting memory operations, the attacker can affect the accesses to some addresses but not others.

With these insights, we present a systematic study of possible windowing capabilities, as shown in Table II. We enumerate all known resources shared between different processes. For each shared resource, we examine existing microarchitectural timing attacks to search for techniques that can slow down other users of this resource and be adjusted to windowing capabilities. To evaluate the effectiveness of these newly discovered windowing capabilities, we quantitatively analyze their latency effect and control granularity with experiments.

Focus of GadgetMeter. In this paper, we focus on one persistent and one volatile windowing capability. For the persistent capability, we choose LLC set eviction, the most powerful capability targeting memory operations. For the volatile capability, we select FP/INT division unit contention, the most powerful capability targeting the processor pipeline. Still, our approach is general and can easily be adapted to other persistent or volatile capabilities.

Among all windowing capabilities targeting memory operations, we select LLC set eviction, which has a larger latency effect and finer control granularity than most other capabilities. One exception is L1D cache bank contention, which can be controlled with even finer granularity. However, it suffers from a limited latency effect (1s of cycles).

Among all windowing capabilities targeting pipeline operations, we choose to focus on division unit contention, which has the largest latency effect (100s of cycles) than any other techniques (1s of cycles) we tested. Contradicting existing work [25], which measures the contention effect on each division instruction to be 1s of cycles, we find that this contention effect can scale to 100s of cycles if the attacker has much more division instructions in the pipeline than the victim thread. Specifically, when the attacker is constantly executing division instructions while the victim only has a few division instructions to execute, the victim's division instructions can be delayed by around 200 cycles on an AMD Zen4 processor and an Intel Skylake processor we tested. We hypothesize that this unusual contention is due to the processor's unfair scheduling policy when two sibling threads have different demands for the FP division unit.

We conducted an experiment to verify our observation and hypothesis. The simplified experiment code is provided in Listing 3a and Listing 3b. In the experiment, an attacker repeatedly executed 7 division instructions with no data dependency. Inside the loop body, we duplicated the 7 division instructions 128 times to obtain a low ratio of control flow logic to overall instructions retired. After the attacker had run for a while, we started a victim thread on a sibling thread, which executed a classical Spectre-V1 gadget with a division-dependent branch. To measure the contention effect on the speculation window size, we gradually delayed the secret leakage by inserting dummy cache hits. We recorded the maximum cache hits inserted before the attack failed.

The results are shown in Figure 2. As can be observed, without SMT contention, the attack fails if the secret leakage is delayed by more than 6 cache hits. However, this number increases to 75 if there's an attacker executing contention workload on the sibling thread. We also conducted the same experiment on the integer division unit, floating point and integer multipliers, and execution ports used by the condition branch. As can be observed from the figure, the latency effect of division unit contention is significantly larger than multiplier and execution ports, indicating that division unit contention is the most powerful.

B. Windowing Strategy

A windowing strategy strives to search for the most optimal attack pattern to optimize the timing condition. An attack pattern is a combination of zero or more unit windowing operations, which are determined by the attacker's windowing capability. Essentially, the windowing strategy acts as a searching algorithm in an optimization problem, where the search space includes all possible attack patterns, and the optimization function is the probability that the leakage instructions are executed before the misspeculated authorizations are resolved if the attack pattern is employed.

TABLE II: Systematic exploration of windowing capabilities and quantitative evaluation based on their latency effect and control granularity.

	Shared Resource	Windowing Capability	Latency Effect on One Target Operation	Control Granularity
Persistent	Cache Set	LLC set eviction [38], [67]	100s of cycles (LLC miss penalty)	64B (typical cache line size)
	TLB	TLB set eviction [27], [31]	10s of cycles (L2 TLB miss penalty)	4KB (typical page size)
	DRAW Row Buffer	DRAW row buffer eviction [49]	10s of cycles (DRAM row conflict penalty)	8KB (DDR4 row size)
	BTB	BTB entry eviction [22], [59]	1s of cycles (BTB miss penalty)	Aliased jmp instructions
	MITE & DSB & LSD	DSB or LSD eviction [21]	1s of cycles (Micro-op delivery path penalty)	Instruction stream
Volatile	Cache Bank	L1D cache bank contention [68]	1s of cycles (double L1D cache latency)	8B (typical cache bank size)
	Memory Bus	Memory bus contention [64]	100s of cycles (memory bus contention penalty)	All accesses on a memory bus
	Memory Controller	Memory controller contention [45], [69]	100s of cycles(memory controller contention penalty)	All accesses through a memory controller
	Execution Ports	Execution port contention [11]	1s of cycles (measured, due to halved instruction throughput)	Instructions using the same port
	FP/INT Division Unit	FP division unit contention [25]	100s of cycles (measured, potentially due to unfair scheduling policy)	Division instructions
	FP/INT Multiplier Unit	Multiplier unit contention [8], [58]	10s of cycles (measured, potentially due to unfair scheduling policy)	Multiplication instructions

```
1:
.rept 128
divd %xmm0, %xmm1
divd %xmm0, %xmm2
...
divd %xmm0, %xmm7
.endr
jmp 1b
```

(a) Attacker

```
if (x < 32./2.){
 y = array1[x];
 idx = buckets[idx];
 ... // N cache hits
 idx = buckets[idx];
 y = y+idx;
 temp&=array2[y*512];
}
```

(b) Victim

Listing 3: Simplified test code to verify the latency effect of SMT FP division unit contention.

A good windowing strategy should achieve a large optimization function and good runtime efficiency. A large optimization function allows the attacker to optimize the timing condition of a gadget and exploit it successfully with a higher success rate. Good runtime efficiency allows the attacker to complete the attack before the environment changes or the attacker is detected.

Windowing strategies used by existing speculative attacks have very limited optimization functions, despite their efficient runtime. They employ an attack pattern consisting of all windowing operations targeting the direct data dependencies of authorization instructions. These strategies can run efficiently as they only rely on simple static analysis. However, they don't perform well on gadgets other than the proof-of-concept ones, as seen in Section II-B.

For a windowing strategy to achieve an optimal optimization function, it has to enumerate through the combinational search space, evaluate each possible attack pattern, and select the most effective one. An example of such a strategy is to construct an attacker-controlled environment that closely resembles the real attack environment, where each attack pattern can be performed, and the one with the highest attack success rate is selected. Despite the optimal attack pattern, this brute-force method takes a long time to run as the pattern space includes all combinations of possible windowing operations. Even worse, performing attacks is time-consuming due to the overhead in setting up the environment, performing attacks, recovering secrets from the side channel, and repeating to compute attack success rates.

Focus of GadgetMeter. In this paper, we propose and simulate an attacker strategy that is **capable of finding an optimal attack pattern with minimal runtime overhead**. To achieve this, we use an analytical model to statistically evaluate and select the most effective attack pattern. More details on this sophisticated attacker strategy, including the specific techniques and algorithms used in the analytical model, will be provided in Section V-B.

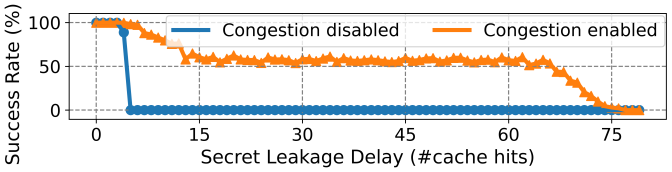
V. Design of GadgetMeter

The goal of GadgetMeter is to quantitatively assess and prioritize speculative gadget exploitability based on timing properties. In Step A (Modeling), we introduce the Directed Acyclic Instruction Graph (iDAG) to analytically model the race between authorization and leakage. In Step B (Attacker Simulation), we propose two algorithms to simulate the powerful windowing capabilities evaluated in Section IV-A, and to select the optimal attack pattern. In Step C (Runtime Testing), we simulate the selected pattern at runtime and measure a "feasibility score" as an intuitive and accurate metric of gadget exploitability.

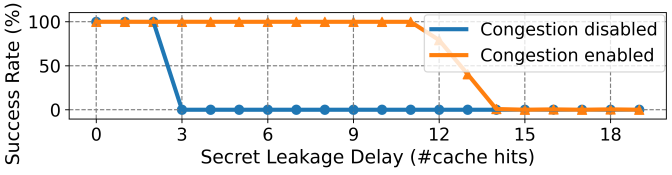
This section will accordingly cover the analytical model, attacker simulation algorithms, GadgetMeter's security metric, and runtime measurement.

A. Directed Acyclic Instruction Graph

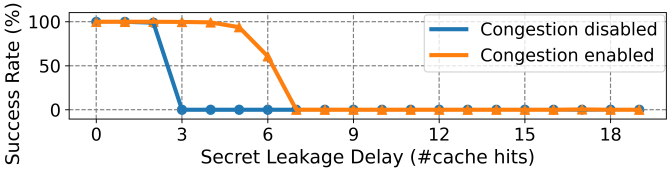
We introduce a Directed Acyclic Instruction Graph (iDAG) to describe the instruction patterns of gadgets. In the graph, each vertex represents an instruction. An edge between two vertices indicates the existence of a timing dependency between the two instructions, i.e., the destination instruction can only be executed after the source instruction is resolved.



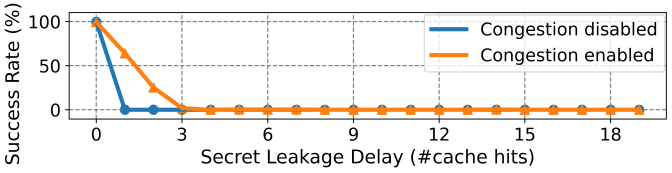
(a) FP DIV contention delays branch resolution by 70 cache hits.



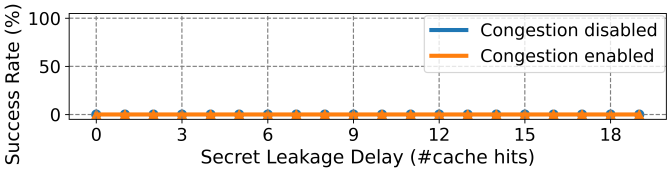
(b) INT DIV contention delays branch resolution by 11 cache hits.



(c) FP MUL contention delays branch resolution by 4 cache hits.



(d) INT MUL contention delays branch resolution by 3 cache hits.



(e) Negligible impact of execution port contention on the branch.

Fig. 2: Experiment results demonstrating the latency effect of different volatile capabilities. We explored contention at FP/INT division and multiplication units and execution ports. As can be observed, SMT contention on the FP/INT division has the most significant latency effect on the victim program.

This timing dependency sources from the read-after-write data dependency between instructions. If there's no path between two vertices, they can execute in parallel. Also, each vertex is associated with a weight, representing the execution latency of the instruction. Therefore, the execution time of a series of dependent instructions can be simplified as the weighted sum of the corresponding path.

Based on this graph, we describe the timing condition of gadgets as a graph property. We use the maximum root-to-sink path sinking at the authorization instruction to represent how long it takes to resolve the misspeculated authorization. We use the maximum root-to-sink path sinking at the leakage instruction to represent how long it takes for the leakage instruction to execute and leak secret. We then model the timing condition of the gadget as the weight comparison of these two max paths:

$$TimingConditionIndex(iDAG) = MaxPathWeight(Auth) - MaxPathWeight(Leak) \quad (1)$$

A larger timing condition index indicates that the gadget is more likely to be exploited successfully since the leakage instructions are left with more time to leak the secret. An example is provided in Figure 3b, which models the timing condition of the gadget in Figure 3a in the absence of an attacker. Note that with dynamic programming, the timing condition index of an iDAG can be computed in *linear time* regarding the number of instructions (vertices) in the iDAG [53].

B. The Analytical Model for Windowing Power

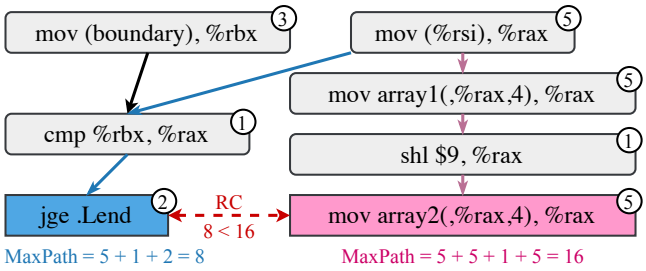
Modelling windowing capability. Based on the Directed Acyclic Instruction Graph, we describe the windowing capability using Algorithm 1. The algorithm models an attacker capable of performing cache line eviction and division unit contention. It takes the raw iDAG of the target gadget and outputs all possible operations that may affect the timing condition. Specifically, the algorithm iterates over each instruction $insn$ in the $iDAG$ and checks its opcode. If the instruction $insn$ involves memory access, the algorithm identifies the target address and includes an eviction operation for the corresponding cache line $targetCacheLine$ within the set of possible actions ops. Likewise, if $insn$ is a division operation, the algorithm adds a division unit contention operation to ops. Notably, this algorithm runs in linear time as it simply iterates over each instruction and performs a constant-time check.

Each operation in the operation set ops in Algorithm 1 can be applied to the raw iDAG and change the execution of some instructions. For a cache line eviction operation, the latency of all instructions that access the affected cache line is increased by 300 cycles. For a division unit contention operation, the latency of all division instructions is increased by 100 cycles. The latency penalty of each operation is approximated based on our pre-processing evaluation in Table II. Such approximation is sufficient for our analytical model to identify the optimal attack pattern, as demonstrated in Figure 4 later in this section.

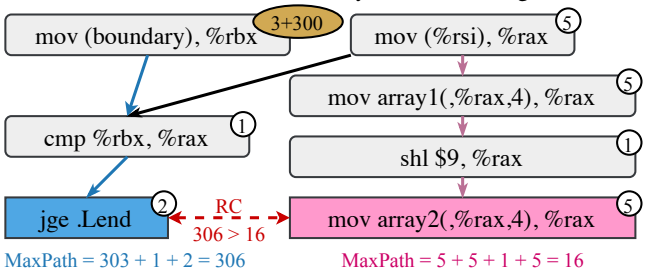
Modelling windowing strategy. We propose a sophisticated windowing strategy that is effective and practically efficient, as described in Algorithm 2. The algorithm iterates over all attack patterns $allPatterns$ in the combinational search space, evaluates their effectiveness with $score$, and selects the most effective one $pattern_{optimal}$. The effectiveness of all attack patterns is evaluated and compared using our analytical model for the timing condition. First, the attack pattern is applied to the raw iDAG of the target gadget, potentially changing the latency (weight) of multiple instructions (vertices). As detailed in the function $ApplyPattern$ in Algorithm 2, we iterate over each operation op in the given pattern $pattern$ and update the latency of target instructions.

```
1  mov (%rsi), %rax
2  mov (boundary), %rbx
3  cmp %rbx, %rax
4  jge .Lend
5  mov array1(,%rax,4), %rax
6  shl $9, %rax
7  mov array2(,%rax,4), %rax
8 .Lend:
```

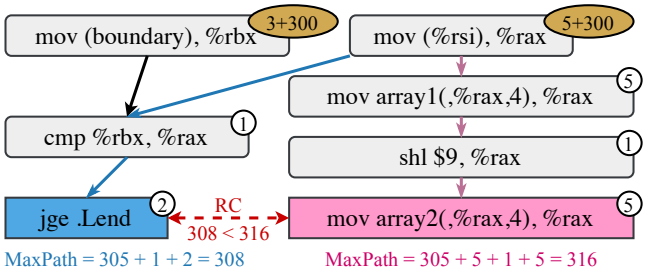
(a) X86 assembly for the PoC Spectre-V1 gadget, except that the attacker-controlled index is in memory instead of a register.



(b) iDAG modeling the timing condition. Timing condition index is $8 - 16 = -8$.



(c) Analytical model for the attack pattern performing cache line eviction of $(boundary)$. Timing condition index increases $(306 - 16 = 290)$ compared with Figure 3b.



(d) Analytical model for the attack pattern performing cache line eviction of both $(boundary)$ and $(\%rsi)$. Timing condition index remains the same $(308 - 316 = -8)$ as Figure 3b.

Fig. 3: An example showing how we use Direct Acyclic Instruction Graph to model the timing condition of a gadget and analyze the effectiveness of different attack patterns. Since attack pattern 1 results in a larger $\Delta_{TimingConditionIndex}$ than attack pattern 2, attack pattern 1 is considered more effective in optimizing the timing condition.

Second, we compute the increase in the Timing Condition Index (TCI) after applying the attack pattern and compare the effectiveness of different attack patterns based on this increase. Specifically, we calculate the new TCI for $iDAG_{new}$ using Equation 1, and determine the difference by subtracting the original TCI_{raw} from the updated TCI_{new}. This increase serves as a metric to evaluate and compare the effectiveness of different attack patterns. Two examples are illustrated in Figure 3c and Figure 3d. In Figure 3c, the attack pattern consists of a single cache line eviction of the address of `boundary`, resulting in a largely increased TCI. For Figure 3d, the attack pattern consists of two cache line evictions, resulting in an unchanged TCI. Therefore, we conclude that the first attack pattern is more effective than the second one as it increases the TCI the most.

Algorithm 2 is guaranteed to identify the optimal attack pattern within our analytical model by exhaustively exploring the entire attack pattern space and selecting the one with the highest effectiveness score. We also demonstrate the algorithm's capability to find the most effective pattern in practice. For instance, for the gadget illustrated in Figure 3, the search space comprises four potential attack patterns, generated by iterating all combination methods of the two cache line eviction operations. We calculate effectiveness scores for each pattern, execute attacks using all four patterns, and record their success rates. To allow for comparison, we standardize both scores and success rates by subtracting the mean and dividing by the standard deviation. Figure 4 reveals a strong correlation between effectiveness scores and attack success rates, demonstrating that Algorithm 2 can reliably identify the optimal attack pattern in practice based on effectiveness scores.

Additionally, Algorithm 2 can maintain practical efficiency. Although the algorithm explores a combinational search space[2], it remains feasible in real-world scenarios for two reasons. First, instead of performing real attacks with each attack pattern as a brute-forth strategy, we evaluate and compare pattern effectiveness with static analysis, which is more efficient and consistent. Second, over 99.28% of the evaluated real-world gadgets contain fewer than five attack operations, with none exceeding ten. This results in a manageable attack pattern search space, allowing our proposed attack strategy to run efficiently in practice.

C. The Security Metric and Runtime Scoring System

Timing feasibility score. To evaluate the exploitability of speculative gadgets based on their timing properties, we propose using timing feasibility as the quantitative security metric. A gadget's timing feasibility refers to the probability that under the assumed windowing power, the leakage instructions can leak secrets transiently before the misspeculated authorization

[2]This exponential complexity is likely inevitable in order to find the optimal attack pattern, because the optimization problem we propose is potentially NP-hard due to its combinational search space and non-trivial objective function. In fact, the 0/1 Knapsack Problem is reducible to a variant of our problem in which we maximize MaxPathWeight(Auth) with a constraint on MaxPathWeight(Leak).

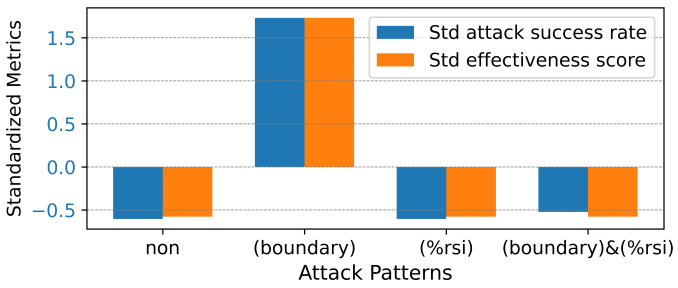


Fig. 4: Correlation between standardized pattern effectiveness scores and standardized real attack success rates for four attack patterns targeting the gadget in Figure 3a. The four attack patterns are combinations of the two windowing operations, namely cache line eviction of $(boundary)$ and $(\%rsi)$.

Algorithm 1 Windowing capability of GadgetMeter

input: a Directed Acyclic Instruction Graph $iDAG$
output: a set of windowing operations ops
$ops \leftarrow \{\}$
for each $insn$ in $iDAG$ **do**
  if $IsMemAccess(insn)$ **then**
    $targetCacheLine \leftarrow GetCacheLine(insn)$
    $newOp \leftarrow$ evict $targetCacheLine$
    $ops \leftarrow ops \cup \{newOp\}$
  end if
  if $IsDiv(insn)$ **then**
    $newOp \leftarrow$ congest division unit
    $ops \leftarrow ops \cup \{newOp\}$
  end if
end for
return ops

Algorithm 2 Windowing strategy of GadgetMeter

input: a Directed Acyclic Instruction Graph $iDAG_{raw}$, a set of windowing operations ops
output: an attack pattern $pattern_{optimal}$
$pattern_{optimal} \leftarrow \{\}$, $score_{max} \leftarrow 0$
for each $pattern$ in $PowerSet(ops)$ **do**
  $iDAG_{new} \leftarrow iDAG_{raw}$
  for each op in $pattern$ **do** ▷ Apply $pattern$ to $iDAG$
    for each $insn$ in $TargetInsnsOf(op)$ **do**
      Update latency of $insn$ in $iDAG_{new}$
    end for
  end for
  $tci_{raw} \leftarrow TimingConditionIndex(iDAG_{raw})$
  $tci_{new} \leftarrow TimingConditionIndex(iDAG_{new})$
  $score \leftarrow tci_{new} - tci_{raw}$ ▷ Impact of $pattern$
  if $score > score_{max}$ **then**
    $pattern_{optimal} \leftarrow pattern$, $score_{max} \leftarrow score$
  end if
end for
return $pattern_{optimal}$

instructions are resolved. Since the timing condition is fundamental to all speculative attacks, a gadget's timing feasibility acts as a universal bottleneck to exploitability.

A runtime approach is essential to measuring the timing feasibility score, even though our proposed Algorithm 2 is capable of assigning effectiveness scores to different attack patterns. This is because while our model computes the location difference between the two latency distributions $(MaxPathWeight(Auth) - MaxPathWeight(Leak))$, which helps distinguish effective attack patterns from ineffective ones (as seen in Section V-B), this location information alone is insufficient to estimate the likelihood of the speculation window being more significant than the disclosure window. The shapes and spreads of the latency distributions, which are necessary for such calculations, can't be accurately captured with static analysis only. Therefore, we opt for a runtime approach to reflect gadgets' exploitability accurately and intuitively.

Runtime scoring system. We derive a gadget's timing feasibility score by measuring and comparing how long it takes for the authorization instructions to resolve (speculation window) and for the leakage instructions to execute (disclosure duration), under the statically selected attack pattern. We avoid deriving the score from real attack success rates to eliminate noises from other steps in an attack such as training or communication. Specifically, to measure the speculation window, we insert timing instructions (e.g., rdtsc) before and after the data dependencies of the authorization instruction. Similarly, we insert timing instructions around the data dependencies of the leakage instruction to measure the disclosure duration. After collecting enough samples from runtime measurement, we derive the timing feasibility by calculating how likely the disclosure duration can fit into the speculation window as follows:

$$\begin{aligned} score &= 10 * P[S > D] \\ &= 10 * \frac{\sum_{i=1}^{n}\sum_{j=1}^{m} 1_{s_i > d_j}}{nm} \end{aligned} \tag{2}$$

where S and D contain all sampled lengths of the speculation window and disclosure duration, respectively. $1_{s_i > d_j}$ is an indicator function that evaluates to 1 if the statement is true and 0 otherwise.

We conducted an experiment to verify that this score aligns well with the actual attack success rate. We used the PoC Spectre-V1 gadget as our test case to reduce noises in other steps of the attack. To examine the accuracy of our method under different timing conditions, we modified the PoC gadget by incrementally adding workloads to the leakage instructions and created 18 test cases (similar to Listing 3b). We performed actual attacks on these gadgets and recorded their success rates. We also evaluated their timing feasibility scores using the method previously described. Figure 5 shows that the timing feasibility scores align well with the actual attack success rates, demonstrating that our proposed security metric can accurately reflect different gadgets' exploitability based on their timing conditions.

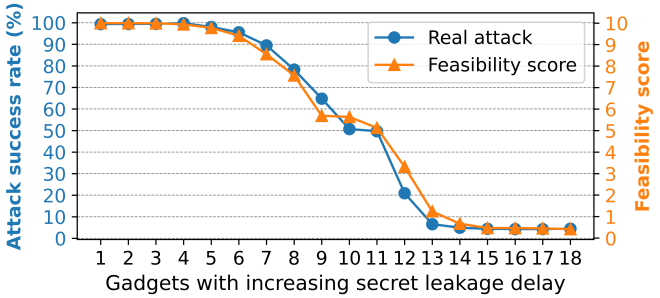


Fig. 5: Correlation between feasibility scores and real attack success rates on 18 gadgets whose exploitability is mainly limited by their timing properties.

VI. GadgetMeter Implementation Details

In Figure 6, we present the workflow of GadgetMeter, which is designed to evaluate the exploitability of gadgets based on their timing conditions. In general, GadgetMeter consists of three steps. In the first step, GadgetMeter constructs a Directed Acyclic Instruction Graph to model the timing condition of the input gadget. In the second step, GadgetMeter simulates an attacker with the windowing power described in Section V-B and selects an effective attack pattern. In the third step, GadgetMeter derives a timing feasibility score through runtime testing. To simplify the implementation, we perform static analysis at the LLVM Intermediate Representation level.

A. Timing Condition Modeling Module To model the timing condition of the input gadget, GadgetMeter builds a Directed Acyclic Instruction Graph by analyzing the use-def chains of IR instructions. In particular, GadgetMeter performs two depth-first searches from the misspeculated branch instruction and the leaking instruction respectively, which are then merged into one DAG. All cycles are eliminated from the graph by duplicating the first nodes causing cycles during the search. To assign weights to each vertex, GadgetMeter evaluates the latencies of each IR instruction by first compiling them into assembly code and querying LLVM's Machine Code Analyzer for latency information.

B. Attacker Simulation Module In this module, GadgetMeter simulates an attacker with the windowing power described in Section V-B. To generate the space of possible attacker patterns, GadgetMeter models an attacker capable of cache line eviction and FP division unit contention as in Algorithm 1. To evaluate the effectiveness of each attack pattern, GadgetMeter applies each pattern to the raw iDAG from the previous step and calculates how the timing condition index is changed, as in Algorithm 2. To figure out whether an instruction will be influenced by a cache eviction operation, GadgetMeter analyzes the alias relationship between different memory locations with LLVM's AliasAnalysis pass. Finally, GadgetMeter selects the attack pattern with the highest analytical score.

C. Runtime Testing Module In this module, GadgetMeter instruments the gadget and profiles it on a real machine. To measure the speculation window, GadgetMeter takes a cycle count using `rdtsc` before and after the data dependencies of the misspeculated branch instruction. The same thing is done to measure the disclosure duration. To simulate the previously selected attack pattern, GadgetMeter inserts a function call to a customized attacker simulator before each measurement. For attack patterns containing cache line evictions, this attacker simulator function performs `clflush` to targeted addresses. For attack patterns containing SMT FP division unit contention, it instantiates a sibling thread that constantly executes division instructions. The measurement is repeated to collect enough samples, with which the timing feasibility score is calculated with Equation 2.

VII. Evaluation

We evaluate GadgetMeter to answer the following two questions:

- How effective is GadgetMeter at evaluating gadgets' exploitability based on their timing condition, compared with existing tools?
- How many of the gadgets with vulnerable information flow in real-world applications are truly exploitable?

In Section VII-A, we compare GadgetMeter against previous approaches with two micro-benchmarks and a macro-benchmark. In Section VII-B, we apply GadgetMeter to evaluate the exploitability of thousands of gadgets discovered by state-of-the-art scanners in six userspace applications and the Linux kernel under our assumed windowing power (cache line eviction and division unit contention).

A. Comparison With Existing Solutions

1) Experiment Setup: We compare GadgetMeter with three prior methods on a variety of datasets.

Baseline methods. We compare GadgetMeter with three different strategies used by existing scanners to evaluate gadgets' timing conditions. To compare with the majority of scanners that consider a gadget exploitable as long as the branch instruction and the leakage instruction can fit into the RoB [28], [33], [48], [50], [57], we use an LLVM pass to count the instructions in between and compare them with the RoB size. To compare with scanners that perform cycle-accurate analysis with the absence of any windowing power [42], we perform attacks on tested gadgets without any cache evicting or SMT congesting behavior and use the attack results to represent their detection results. To compare with scanners that consider every memory-dependent branch vulnerable [20], [29], [63], we use an LLVM pass to examine the data dependencies of target branches.

Evaluation datasets. The evaluation is conducted on three datasets. We collect Kocher's 15 Spectre examples as a plausible microbenchmark for the basic detection capabilities. Still, Kocher's dataset is simple as all gadgets include a branch that compares an attacker-controlled index with a boundary value in memory. To evaluate how the baseline methods perform on more complex gadgets, we extend Kocher's dataset with another 15 gadgets, which have complex data dependencies and resemble gadgets we observe in real-world programs. We also collect 15 gadgets in LibYAML, a C library for parsing

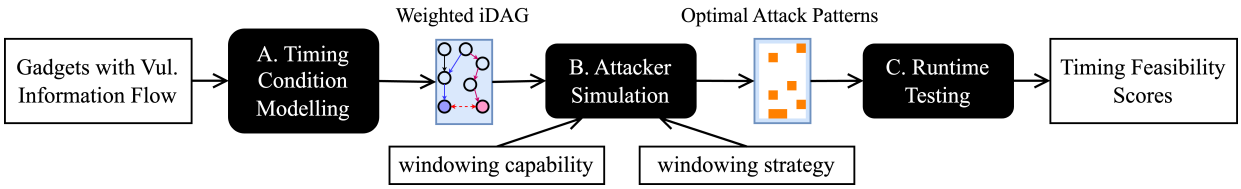


Fig. 6: GadgetMeter workflow, which takes in a gadget with vulnerable information flow, evaluates its timing condition under a simulated attacker, and presents a timing feasibility score as its exploitability.

and emitting YAML; each gadget possesses a vulnerable information flow as detected by state-of-the-art scanner SpecFuzz.

Evaluation metrics. For the two micro-benchmarks, we perform attacks on each gadget to verify their exploitability and use the attack results as the ground truth. For LibYAML, we verify the original gadgets, construct test gadgets with similar data dependency structures, and use the attack results on these test gadgets as the ground truth. Next, we calculate the precision, recall, and accuracy to quantify the effectiveness of our proposed approach and baseline methods. The precision is calculated as $P\% = \frac{TP}{TP+FP}$. The recall is calculated as $R\% = \frac{TP}{TP+FN}$. The accuracy is calculated as $A\% = \frac{TP+TN}{TP+TN+FP+FN}$.

Configuration. The experiments are conducted natively on a server with Intel(R) Xeon(R) Silver 4114 CPU and 187G of RAM and running Linux 5.15.

2) Experiment Results: As can be observed from Table III, GadgetMeter performs better than all three existing solutions in precision, recall, and accuracy.

Kocher's micro-benchmark. According to our attack results, all 15 gadgets are exploitable. This is because they share the same data dependency structure: they take in an attacker-controlled index, perform a sanity check using some boundary value in memory, and access memory using the controlled index. To exploit each gadget, the attacker can flush the boundary value used for the sanity check out of the cache. This action delays the branch resolution by hundreds of cycles.

Due to the simplicity of this micro-benchmark, all solutions other than the cycle-accurate simulation perform well on this dataset. The cycle-accurate simulation strategy falsely classifies all 15 gadgets as unexploitable due to the absence of windowing power simulation. This strategy doesn't perform cache eviction on the boundary value used by the branch. As a result, the speculation window isn't large enough for the secret to be transmitted, as the boundary value tends to reside in the cache for frequent accesses.

GadgetMeter's micro-benchmark. To establish the ground truth for this extended micro-benchmark, we try out all possible attack patterns on each gadget and consider a gadget exploitable if any of the patterns succeed. According to the attack results, 7 gadgets are exploitable, while 8 are not.

Due to the complexity of this micro-benchmark, the three baseline methods misclassify many gadgets. The instruction counting strategy only considers the instruction-level distance between the branch and the fault but ignores their timing, thus falsely classifying 8 unexploitable gadgets as vulnerable. The cycle-accurate simulation strategy fails to identify 6 vulnerable gadgets, which are exploitable only if the attacker performs cache eviction and SMT contention actions. The memory access detection strategy misclassifies 5 unexploitable gadgets as vulnerable. The misclassifications result from the fact that not all gadgets with a memory-dependent branch can be exploited, as seen in Listing 2b and Listing 2c. This strategy also misclassifies 3 vulnerable gadgets as unexploitable due to their lack of memory-dependent branch. In contrast, GadgetMeter correctly classifies the exploitability of every gadget.

JSMN. To establish the ground truth for this macro-benchmark, we first construct a series of test gadgets sharing the same data dependency structures as the raw gadgets in the program. Next, similar to what we did to the micro-benchmarks, we enumerate all possible attack patterns on each test gadget. We consider a raw gadget exploitable if the corresponding test gadget can be exploited by any of the attack patterns. According to the attack results, 7 gadgets are exploitable, while 8 are not.

Similar to GadgetMeter's micro-benchmark, all three baseline methods misclassify a significant portion of the gadgets, while GadgetMeter correctly classifies all of them. One notable difference is that the memory access detection strategy correctly classifies all exploitable gadgets, achieving 100% recall. This is because of the instruction and data dependency patterns of JSMN, making a memory-dependent branch a necessary condition for an exploitable gadget.

B. Evaluting Gadgets in Real-World Programs

To demonstrate GadgetMeter's capability to evaluate gadgets' exploitability in large-scale applications, we apply GadgetMeter on real-world gadgets with vulnerable information flow identified by two state-of-the-art scanners SpecFuzz [48] and Kasper [33].

Gadget collection. We configure SpecFuzz and Kasper according to their evaluation setup. We run SpecFuzz on each application for one hour to collect gadgets as pairs of (mispredicted branch, fault). For gadgets with nested branch mispredictions, we evaluate the timing feasibility score for each pair of branch instruction and fault instruction. Then, we use the lowest score to represent the score for this gadget because successfully exploiting a nested gadget requires the secret to be transmitted before any of those mispredicted branches resolve. We also run Kasper on the Linux kernel for 6 hours and collect gadgets as tuples of (mispredicted branch, secret

TABLE III: Evaluation results of three baseline methods and GadgetMeter on three datasets. The leftmost column is the ground truth established based on attack results. (P: positive; N: negative; FP: false positive; FN: false negative; P%: precision; R%: recall; A%: accuracy)

Datasets			Instruction Counting					Cycle-Accurate Sim					Mem Access Detection					GadgetMeter				
Name	P	N	FP	FN	P%	R%	A%	FP	FN	P%	R%	A%	FP	FN	P%	R%	A%	FP	FN	P%	R%	A%
Kocher's	15	0	0	0	100	100	100	0	15	NA	0.0	0.0	0	0	100	100	100	0	0	100	100	100
GMeter's	7	8	8	0	46.7	100	46.7	0	6	100	14.3	60.0	5	3	44.4	57.1	46.7	0	0	100	100	100
JSMN	7	8	8	0	46.7	100	46.7	0	4	100	42.9	73.3	8	0	46.7	100	46.7	0	0	100	100	100
Total	29	16	16	0	64.4	100	64.4	0	25	100	13.8	44.4	13	3	66.7	90.0	64.4	0	0	100	100	100

TABLE IV: Timing feasibility scores for gadgets in six security-centric userspace applications and the Linux kernel, where GadgetMeter identifies 503 unexplotiable, 3390 mildly exploitable, and 852 exploitable gadgets.

Score	0	1	2	3	4	5	6	7	8	9	10
Brotli	154	66	74	35	16	17	18	13	10	69	249
HTTP	3	1	1	3	0	1	1	0	1	1	2
JSMN	8	0	0	0	1	1	0	0	0	1	4
LibHTP	36	25	29	41	18	8	7	8	0	9	67
LibYAML	9	12	26	15	0	3	1	3	4	46	65
OpenSSL	261	200	229	364	215	82	58	33	11	61	236
Linux	32	36	92	157	304	345	237	111	66	204	229
Total	503	3390									852

access, secret leakage). We target the mispredicted branch and the secret access and evaluate their timing feasibility score. To achieve a conservative result, we don't include the secret leakage instruction because the leakage instruction does not have to be fully executed (transiently) to transmit the secret.

Configuration. For the six userspace applications, the experiments are conducted natively on a server with Intel(R) Xeon(R) Silver 4114 CPU and 187G of RAM and running Linux 5.15. For the Linux kernel, the experiments are conducted on an Intel(R) Xeon(R) Gold 5318Y CPU with 252G of RAM, where the GadgetMeter-instrumented kernel (v5.12) runs as a guest VM on a host running Ubuntu 22.04.2 LTS (kernel 5.15.0-73-generic).

Results. The experimental results are summarized in Table IV. Of all gadgets analyzed, 503 received a timing feasibility score of 0, indicating they are unexploitable under our assumed windowing power. This includes 48% with no memory access or division instructions, and 52% affected by shared memory and division instructions that limit exploitation. In contrast, 852 gadgets were scored 10, signifying high vulnerability, primarily due to cache line evictions, with four caused by division unit contention. The remaining gadgets were scored between 1 and 9, suggesting mild exploitability with varying success rates. We present three case studies in Section IX and Appendix A.

To further verify the evaluation results, we introduce a ground truth analysis with cross-validation on the three baseline methods used in Section VII-A. We manually verified any unmatched result to derive the ground truth. As shown in Table V, GadgetMeter outperforms prior methods in both under/over-approximation rates. First, we verified that all of

TABLE V: Under/over-approximations resulted from GadgetMeter and prior approaches in the large-scale experiment. An evaluation score is classified as an under/over-estimation if it's smaller/larger than the ground truth by 5 out of 10.

	Under-Approx.	**Over-Approx.**
Instruction Counting	0%	51.10%
Cycle-Accurate simulation	16.44%	0%
Mem Access Detection	9.63%	23.60%
GadgetMeter	0.67%	0.63%

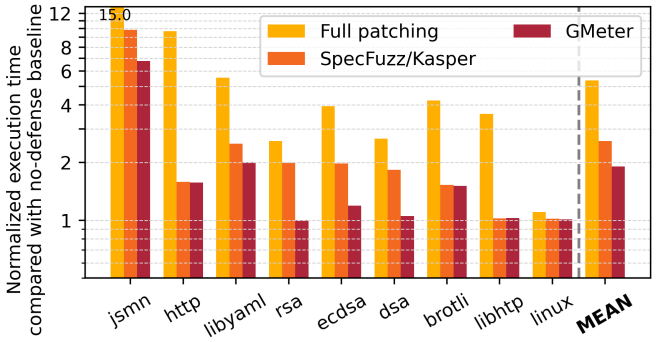


Fig. 7: The performance overhead after patching w.r.t the execution time with no defense added.

the instruction counting method's over-estimations are due to its inability to model the timing condition of gadgets. Second, we verified that all of the cycle-accurate simulation method's under-estimations are due to its inability to simulate even the most naive attacker (as in Lising 2a). Third, we verified that all of the memory access detection method's over-estimations are due to its naive windowing strategy (as in Listing 2b and Listing 2c). At the same time, all of the under-estimations are due to its limited view of what windowing capabilities comprise (as in Listing 2d).

Also, we identify two sources of over/under-approximations in GadgetMeter, with a smaller ratio compared with the existing three approaches. First, we verified that GadgetMeter's over-estimations are due to our conservative instrumentation method in Step C runtime testing. We also verified that GadgetMeter's underestimation is due to inaccuracy in the static analysis passes used in Steps 1 and 2. Nevertheless, the misclassification rate remains acceptable. We discuss possible solutions in Section VIII.

Finally, we evaluate the performance gained by eliminating patches for the unexploitable gadgets identified by Gadget-

Meter. Specifically, we utilize the serialization tool provided by SpecFuzz to patch programs by selectively serializing gadgets with LFENCE. We use the same benchmarks as SpecFuzz for all applications. For the Linux kernel, we use sysbench [36], a widely adopted system benchmark tool. We patch all gadgets reported by state-of-the-art scanners SpecFuzz and Kasper. For GadgetMeter, we only patch exploitable gadgets with a non-zero score. For the full-patching baseline, we patch every conditional branch.

As shown in Figure 7, GadgetMeter reduces the patching overhead by up to 31.2% on JSMN and 44.1% on OpenSSL. On average, GadgetMeter reduces the patching overhead by 20.66% compared with state-of-the-art scanners' patching and 59.80% with full hardening, as a result of 29.01% less fences compared with the two baseline scanners. After an investigation with *perf*, we noticed that the speedup is mainly due to the eliminated patching for false positives on program hot paths. For example, in JSMN, we observed that most of the improvement is due to removing a single fence in a hot code region responsible for 98.7% of executed instructions. Similar trends are observed in OpenSSL and LibYAML, demonstrating the performance gains achievable with a high-precision gadget scanner, especially for gadgets on hot paths.

VIII. Discussion

In this section, we discuss the limitations we have discovered in the current implementation of GadgetMeter. We also highlight potential ways to overcome these limitations.

Dependency analysis. During the cross-validation for the large-scale experiment, we discovered that inaccuracy in the compiler analysis tool that GadgetMeter relies on may result in misclassifications. In Steps A and B, GadgetMeter relies on LLVM Dependencey Analysis [3] to construct an iDAG for each gadget and select the optimal attack pattern. Its limited precision, especially when dealing with pointer alias analysis, may result in an inaccurate iDAG and a suboptimal attack pattern, leading to under-approximations. In Step C, to deal with the unclear dependencies, GadgetMeter conservatively adjusts the window measurement to include (or exclude) unclear dependencies for branches (or faults). This might lead to over-approximations. To fix this, we can leverage more advanced dependency analysis tools, such as the dynamic alias analysis error detector NeoGoby [62].

Hardware dependency. Although GadgetMeter's evaluation results are specific to hardware, our toolchain can be operated on different hardware with ease. In Steps B and C, GadgetMeter obtains instruction latencies for different microarchitectures using LLVM MCA [4]. While LLVM MCA offers extensive support, alternative tools [7], [24] exist to measure latencies on arbitrary processors. Moreover, Step C has low overhead, as we only perform runtime testing on one optimal attack pattern for each gadget.

Windowing capability modeling. Our prototype focuses on cache line eviction and division unit contention, *but can be easily adapted to other capabilities*. We define each windowing capability using two rules: a) how it differentiates between instructions (control granularity) and b) its impact on instruction latencies (latency effect). Each capability is simulated through a runtime function activated before executing each target gadget. Adapting the prototype for different attacker capabilities is straightforward, provided they can be defined by the two rules and simulated through instruction execution. Our prototype demonstrates this by simulating one volatile and one persistent capability. We also extend our prototype to simulate multiplier unit contention and successfully observe two more gadgets enabled in OpenSSL.

```
1  typedef struct BrotliTransforms { ...
2    uint32_t num_transforms;
3    const uint8_t* transforms; ...
4  } BrotliTransforms;
5  BrotliDecoderErrorCode ProcessCommandsInternal
       (...) { ...
6    if (transform_idx <
       transforms->num_transforms) {
7      len = BrotliTransformDictionaryWord(...,
       transforms, transform_idx); ...
8  }
9  #define BROTLI_TRANSFORM_SUFFIX_ID(T, I) \
10   ((T)->transforms[((I) * 3) + 2])
11 #define BROTLI_TRANSFORM_SUFFIX(T, I) \
12   (&(T)->prefix_suffix[ \
13   (T)->prefix_suffix_map[ \
14   BROTLI_TRANSFORM_SUFFIX_ID(T, I)]])
15 int BrotliTransformDictionaryWord(...,
16   const BrotliTransforms* transforms,
17   int transform_idx) {
18   const uint8_t* suffix =
       BROTLI_TRANSFORM_SUFFIX(transforms,
       transform_idx);
19   ...
```

Listing 4: An unexploitable gadget in Brotli.

IX. Case Study – Unexploitable Gadget in Brotli

Brotli [10] is a generic-purpose lossless compression program. SpecFuzz reports a vulnerable information flow introduced by a gadget in function *ProcessCommandsInternal* and *BrotliTransformDictionaryWord*, as presented in Listing 4. SpecTaint [50] also detects this gadget and uses this gadget as a case study to demonstrate its capability. The vulnerable branch is on line 6, and the macro on line 10 performs the vulnerable memory access. The program first checks whether the *transfrom_idx* propagated from user inputs is within a boundary value *transforms* $\rightarrow$ *num_transforms*. If the index is within the boundary, *transform_idx* and *transforms* are then passed to *BrotliTransformDictionaryWord*, which performs a series of indirect memory accesses from line 9 to line 14.

SpecTaint considered this gadget as exploitable because a cache miss on the boundary value *transforms* $\rightarrow$ *num_transforms* could delay the branch resolution by hundreds of cycles, thus opening up the speculation window for leaking secrets. However, GadgetMeter scored this gadget at 0, suggesting that the gadget is unexploitable. After careful examination, we noticed that due to the definition of the

structure *BrotliTransforms* on line 1 to line 4, the boundary value *transforms* $\rightarrow$ *num_transforms* is co-located on the same cache line with the base array pointer *transforms* $\rightarrow$ *transforms*, which is used on line 10 to access an out-of-bound memory secret. Therefore, a cache line eviction of the boundary value *transforms* $\rightarrow$ *num_transforms* will also halt the secret leaking, thus failing the attack. We verified that this gadget was unexploitable in practice, even if the boundary value was flushed out of the cache.

We present two more case studies in Appendix A: one identifies a gadget in Linux with low exploitability, and the other finds an exploitable gadget in OpenSSL.

X. Conclusion

This paper presents GadgetMeter, a framework that improves gadget exploitability assessment using precise timing analysis. By employing a Directed Acyclic Instruction Graph and combining static and runtime analysis, GadgetMeter accurately evaluates and prioritizes vulnerabilities. Benchmarks show that GadgetMeter outperforms existing scanners in accuracy. Overall, GadgetMeter enhances mitigation strategies with minimal performance impact while improving security.

Acknowledgments

This work was generously supported by NSFC (U24A6009), Beijing Natural Science Foundation (L247013), BNRist, the PRISM Research Center, a JUMP Center co-sponsored by SRC and DARPA, ONR, and NSF (1942218).

References

[1] "1528 - speculative execution, variant 4: speculative store bypass - project-zero." [Online]. Available: https://bugs.chromium.org/p/project-zero/issues/detail?id=1528

[2] "Linux-Kernel Archive: Performance Regression in Linux Kernel 5.19." [Online]. Available: https://lkml.iu.edu/hypermail/linux/kernel/2209.1/02248.html

[3] "LLVM: lib/Analysis/DependenceAnalysis.cpp Source File." [Online]. Available: https://llvm.org/doxygen/DependenceAnalysis_8cpp_source.html

[4] "llvm-mca - LLVM Machine Code Analyzer LLVM 20.0.0git documentation." [Online]. Available: https://llvm.org/docs/CommandGuide/llvm-mca.html

[5] "Paul Kocher: Spectre Mitigations in Microsoft's C/C++ Compiler." [Online]. Available: https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html

[6] "straight-line speculation." [Online]. Available: https://smist08.wordpress.com/tag/straight-line-speculation/

[7] A. Abel and J. Reineke, "uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Apr. 2019, pp. 673–686, arXiv:1810.04610 [cs]. [Online]. Available: http://arxiv.org/abs/1810.04610

[8] O. Aciicmez and J.-P. Seifert, "Cheap Hardware Parallelism Implies Cheap Security," in *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2007)*, Sep. 2007, pp. 80–91. [Online]. Available: https://ieeexplore.ieee.org/document/4318988

[9] P. Aimoniotis, C. Sakalis, M. Sjalander, and S. Kaxiras, "Reorder Buffer Contention: A Forward Speculative Interference Attack for Speculation Invariant Instructions," *IEEE Computer Architecture Letters*, vol. 20, no. 2, pp. 162–165, Jul. 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9591413/

[10] J. Alakuijala, A. Farruggia, P. Ferragina, E. Kliuchnikov, R. Obryk, Z. Szabadka, and L. Vandevenne, "Brotli: A general-purpose data compressor," *ACM Transactions on Information Systems (TOIS)*, vol. 37, no. 1, pp. 1–30, 2018.

[11] A. C. Aldaya, B. B. Brumley, S. u. Hassan, C. P. García, and N. Tuveri, "Port Contention for Fun and Profit," 2018, publication info: Published elsewhere. To appear in the Proceedings of the IEEE Symposium on Security & Privacy, May 2019. [Online]. Available: https://eprint.iacr.org/2018/1060

[12] Anonymous, "Project Zero: Reading privileged memory with a side-channel," Jan. 2018. [Online]. Available: https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html

[13] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "SMoTherSpectre: Exploiting Speculative Execution through Port Contention," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, Nov. 2019, pp. 785–800. [Online]. Available: https://dl.acm.org/doi/10.1145/3319535.3363194

[14] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. v. Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A Systematic Evaluation of Transient Execution Attacks and Defenses," 2019, pp. 249–266. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/canella

[15] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom, "Fallout: Leaking Data on Meltdown-resistant CPUs," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, Nov. 2019, pp. 769–784. [Online]. Available: https://dl.acm.org/doi/10.1145/3319535.3363219

[16] S. Cauligi, C. Disselkoen, D. Moghimi, G. Barthe, and D. Stefan, "SoK: Practical Foundations for Software Spectre Defenses," in *2022 IEEE Symposium on Security and Privacy (SP)*, May 2022, pp. 666–680, iSSN: 2375-1207.

[17] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T.-H. Lai, "SgxPectre: Stealing Intel Secrets From SGX Enclaves via Speculative Execution," *IEEE Security & Privacy*, vol. 18, no. 3, pp. 28–37, May 2020. [Online]. Available: https://ieeexplore.ieee.org/document/8967194/

[18] Y. Cohen, K. S. Tharayil, A. Haenel, D. Genkin, A. D. Keromytis, Y. Oren, and Y. Yarom, "HammerScope: Observing DRAM Power Consumption Using Rowhammer," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. Los Angeles CA USA: ACM, Nov. 2022, pp. 547–561. [Online]. Available: https://dl.acm.org/doi/10.1145/3548606.3560688

[19] I. Corporation, "Intelő 64 and ia-32 architectures optimization reference manual volume 1," 2016. [Online]. Available: https://www.intel.com/content/www/us/en/content-details/671488/intel-64-and-ia-32-architectures-optimization-reference-manual-volume-1.html

[20] L.-A. Daniel, S. Bardin, and T. Rezk, "Hunting the Haunter Efficient Relational Symbolic Execution for Spectre with Haunted RelSE," in *Proceedings 2021 Network and Distributed System Security Symposium*. Virtual: Internet Society, 2021. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/ndss2021_4B-4_24286_paper.pdf

[21] S. Deng, B. Huang, and J. Szefer, "Leaky frontends: Security vulnerabilities in processor frontends," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 53–66.

[22] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over ASLR: Attacking branch predictors to bypass ASLR," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Taipei, Taiwan: IEEE, Oct. 2016, pp. 1–13. [Online]. Available: http://ieeexplore.ieee.org/document/7783743/

[23] L. Fiolhais and L. Sousa, "Transient-Execution Attacks: A Computer Architect Perspective," *ACM Computing Surveys*, vol. 56, no. 3, pp. 74:1–74:38, Oct. 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/3603619

[24] A. Fog, "Optimizing subroutines in assembly language."

[25] J. Fustos, M. Bechtel, and H. Yun, "SpectreRewind: Leaking Secrets to Past Instructions," in *Proceedings of the 4th ACM Workshop on Attacks and Solutions in Hardware Security*, ser. ASHES'20. New York, NY,

USA: Association for Computing Machinery, Nov. 2020, pp. 117–126. [Online]. Available: https://dl.acm.org/doi/10.1145/3411504.3421216

[26] D. Goodin, "New Spectre attack once again sends Intel and AMD scrambling for a fix - Ars Technica." [Online]. Available: https://arstechnica.com/gadgets/2021/05/new-spectre-attack-once-again-sends-intel-and-amd-scrambling-for-a-fix/

[27] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks."

[28] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "SPECTECTOR: Principled Detection of Speculative Information Flows," Jul. 2019, arXiv:1812.08639 [cs]. [Online]. Available: http://arxiv.org/abs/1812.08639

[29] S. Guo, Y. Chen, P. Li, Y. Cheng, H. Wang, M. Wu, and Z. Zuo, "SpecuSym: speculative symbolic execution for cache timing leak detection," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, Oct. 2020, pp. 1235–1247. [Online]. Available: https://doi.org/10.1145/3377811.3380428

[30] E. Haletky, *VMware ESX and ESXi in the Enterprise: Planning Deployment of Virtualization Servers*. Pearson Education, 2011.

[31] R. Hund, C. Willems, and T. Holz, "Practical Timing Side Channel Attacks against Kernel Space ASLR," in *2013 IEEE Symposium on Security and Privacy*. Berkeley, CA: IEEE, May 2013, pp. 191–205. [Online]. Available: http://ieeexplore.ieee.org/document/6547110/

[32] M. H. Islam Chowdhuryy, H. Liu, and F. Yao, "BranchSpec: Information Leakage Attacks Exploiting Speculative Branch Instruction Executions," in *2020 IEEE 38th International Conference on Computer Design (ICCD)*, Oct. 2020, pp. 529–536, iSSN: 2576-6996. [Online]. Available: https://ieeexplore.ieee.org/document/9283585

[33] B. Johannesmeyer, J. Koschel, K. Razavi, H. Bos, and C. Giuffrida, "Kasper: Scanning for Generalized Transient Execution Gadgets in the Linux Kernel," in *Proceedings 2022 Network and Distributed System Security Symposium*. San Diego, CA, USA: Internet Society, 2022. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2022-221-paper.pdf

[34] M. F. A. Kadir, J. K. Wong, F. Ab Wahab, A. F. A. A. Bharun, M. A. Mohamed, and A. H. Zakaria, "Retpoline technique for mitigating spectre attack," in *2019 6th International Conference on Electrical and Electronics Engineering (ICEEE)*. IEEE, 2019, pp. 96–101.

[35] 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 *2019 IEEE Symposium on Security and Privacy (SP)*, May 2019, pp. 1–19, iSSN: 2375-1207.

[36] A. Kopytov, "akopytov/sysbench," Oct. 2024, original-date: 2015-03-07T08:27:40Z. [Online]. Available: https://github.com/akopytov/sysbench

[37] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre Returns! Speculation Attacks using the Return Stack Buffer," 2018. [Online]. Available: https://www.usenix.org/conference/woot18/presentation/koruyeh

[38] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks are Practical," in *2015 IEEE Symposium on Security and Privacy*. San Jose, CA: IEEE, May 2015, pp. 605–622. [Online]. Available: https://ieeexplore.ieee.org/document/7163050/

[39] G. Maisuradze and C. Rossow, "ret2spec: Speculative Execution Using Return Stack Buffers," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, Oct. 2018, pp. 2109–2122. [Online]. Available: https://dl.acm.org/doi/10.1145/3243734.3243761

[40] A. Mambretti, M. Neugschwandtner, A. Sorniotti, E. Kirda, W. Robertson, and A. Kurmus, "Speculator: a tool to analyze speculative execution attacks and mitigations," in *Proceedings of the 35th Annual Computer Security Applications Conference*. San Juan Puerto Rico USA: ACM, Dec. 2019, pp. 747–761. [Online]. Available: https://dl.acm.org/doi/10.1145/3359789.3359837

[41] A. Mambretti, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, and A. Kurmus, "Two methods for exploiting speculative control ow hijacks."

[42] Z. McKevitt, A. Trivedi, and T. S. Lehman, "SpecCheck: A Tool for Systematic Identification of Vulnerable Transient Execution in gem5," in *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2023, pp. 265–278. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10364588

[43] J. D. Meulemeester, A. Purnal, L. Wouters, A. Beckers, and I. Verbauwhede, "{SpectrEM}: Exploiting Electromagnetic Emanations During Transient Execution," 2023, pp. 6293–6310. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/de-meulemeester

[44] A. Milburn, K. Sun, and H. Kawakami, "You Cannot Always Win the Race: Analyzing the LFENCE/JMP Mitigation for Branch Target Injection," Mar. 2022, arXiv:2203.04277 [cs]. [Online]. Available: http://arxiv.org/abs/2203.04277

[45] T. Moscibroda and O. Mutlu, "Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems."

[46] O. Oleksenko, M. Guarnieri, B. Köpf, and M. Silberstein, "Hide and Seek with Spectres: Efficient discovery of speculative information leaks with random testing," Jan. 2023, arXiv:2301.07642 [cs]. [Online]. Available: http://arxiv.org/abs/2301.07642

[47] O. Oleksenko, B. Trach, T. Reiher, M. Silberstein, and C. Fetzer, "You Shall Not Bypass: Employing data dependencies to prevent Bounds Check Bypass," Oct. 2018, arXiv:1805.08506 [cs]. [Online]. Available: http://arxiv.org/abs/1805.08506

[48] O. Oleksenko, B. Trach, M. Silberstein, and C. Fetzer, "Bringing Spectre-type vulnerabilities to the surface."

[49] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks."

[50] Z. Qi, Q. Feng, Y. Cheng, M. Yan, P. Li, H. Yin, and T. Wei, "SpecTaint: Speculative Taint Analysis for Discovering Spectre Gadgets," in *Proceedings 2021 Network and Distributed System Security Symposium*. Virtual: Internet Society, 2021. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/ndss2021_4B-5_24466_paper.pdf

[51] A. Randal, "This is How You Lose the Transient Execution War," Sep. 2023, arXiv:2309.03376 [cs]. [Online]. Available: http://arxiv.org/abs/2309.03376

[52] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss, "NetSpectre: Read Arbitrary Memory over Network," in *Computer Security ESORICS 2019*, K. Sako, S. Schneider, and P. Y. A. Ryan, Eds. Cham: Springer International Publishing, 2019, vol. 11735, pp. 279–299, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-030-29959-0_14

[53] R. Sedgewick and K. Wayne, *Algorithms*. Addison-Wesley Professional, Feb. 2011, google-Books-ID: idUdqdDXqnAC.

[54] J. Szefer, "Survey of Microarchitectural Side and Covert Channels, Attacks, and Defenses," *Journal of Hardware and Systems Security*, vol. 3, no. 3, pp. 219–234, Sep. 2019. [Online]. Available: http://link.springer.com/10.1007/s41635-018-0046-1

[55] Y. Tobah, A. Kwong, I. Kang, D. Genkin, and K. G. Shin, "SpecHammer: Combining Spectre and Rowhammer for New Speculative Attacks," in *2022 IEEE Symposium on Security and Privacy (SP)*, May 2022, pp. 681–698, iSSN: 2375-1207. [Online]. Available: https://ieeexplore.ieee.org/document/9833802

[56] D. Trujillo, J. Wikner, and K. Razavi, "InceptIon: Exposing New Attack Surfaces with Training in Transient Execution."

[57] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury, "oo7: Low-overhead Defense against Spectre Attacks via Program Analysis," Nov. 2019, arXiv:1807.05843 [cs]. [Online]. Available: http://arxiv.org/abs/1807.05843

[58] Z. Wang and R. B. Lee, "Covert and Side Channels Due to Processor Architecture," in *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, Dec. 2006, pp. 473–482, iSSN: 1063-9527. [Online]. Available: https://ieeexplore.ieee.org/document/4041191

[59] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci, "NDA: Preventing Speculative Execution Attacks at Their Source," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. Columbus OH USA: ACM, Oct. 2019, pp. 572–586. [Online]. Available: https://dl.acm.org/doi/10.1145/3352460.3358306

[60] J. Wikner and K. Razavi, "{RETBLEED}: Arbitrary Speculative Code Execution with Return Instructions," 2022, pp. 3825–3842. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/wikner

[61] J. Wikner, D. Trujillo, and K. Razavi, "Phantom: Exploiting Decoder-detectable Mispredictions," in *56th Annual IEEE/ACM International Symposium on Microarchitecture*. Toronto ON Canada: ACM, Oct.

```
1  #define list_for_each_entry(pos,head,member)
2      for (pos = list_first_entry(head,typeof(*
       pos),member); \
3      &pos->member != head ; \
4      pos = list_next_entry(pos,member))
5  static int flock_lock_inode(struct inode *inode
       , struct file_lock *request) {
6    struct file_lock *fl;
7    struct file_lock_context *ctx =
       locks_get_lock_context(inode, request->
       fl_type);
8    ...
9    list_for_each_entry(fl, ctx->flc_flock ,
       fl_list) {
10     if (request->fl_file != fl->fl_file)
11       continue;
12   ...
```

Listing 5: A gadget in Linux kernel with relatively low exploitability.

```
1  BIGNUM *bn_wexpand(BIGNUM *a, int words) {
2      return (words <= a->dmax) ? a : bn_expand2(
       a, words);
3  }
4  static BIGNUM *bin2bn(const unsigned char *s,
       int len, BIGNUM *ret,...) { ...
5      n = ((len - 1) / BN_BYTES) + 1;
6      if (!ossl_assert(bn_wexpand(ret, (int)n )
       != NULL)) { ...
7          return NULL;
8      } ...
9      for (i = 0; n-- > 0 ; i++) { ...
10         ret->d[i] = l;
11         ...
```

Listing 6: An exploitable gadget in OpenSSL.

2023, pp. 49–61. [Online]. Available: https://dl.acm.org/doi/10.1145/3613424.3614275

[62] J. Wu, G. Hu, Y. Tang, and J. Yang, "Effective dynamic detection of alias analysis errors," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. Saint Petersburg Russia: ACM, Aug. 2013, pp. 279–289. [Online]. Available: https://dl.acm.org/doi/10.1145/2491411.2491439

[63] M. Wu and C. Wang, "Abstract interpretation under speculative execution," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Phoenix AZ USA: ACM, Jun. 2019, pp. 802–815. [Online]. Available: https://dl.acm.org/doi/10.1145/3314221.3314647

[64] Z. Wu, Z. Xu, and H. Wang, "Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud."

[65] Y. Xiao, Y. Zhang, and R. Teodorescu, "SPEECHMINER: A Framework for Investigating and Measuring Speculative Execution Vulnerabilities," in *Proceedings 2020 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2020. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2020/02/23105.pdf

[66] W. Xiong and J. Szefer, "Survey of Transient Execution Attacks and Their Mitigations," *ACM Computing Surveys*, vol. 54, no. 3, pp. 54:1–54:36, May 2021. [Online]. Available: https://doi.org/10.1145/3442479

[67] Y. Yarom and K. Falkner, "Flush+Reload: A High Resolution, Low Noise, L3 Cache Side-Channel Attack."

[68] Y. Yarom, D. Genkin, and N. Heninger, "CacheBleed: A Timing Attack on OpenSSL Constant Time RSA," 2016, publication info: Preprint. MINOR revision. [Online]. Available: https://eprint.iacr.org/2016/224

[69] T. Zhang, Y. Zhang, and R. B. Lee, "Memory DoS Attacks in Multi-tenant Clouds: Severity and Mitigation," Oct. 2017, arXiv:1603.03404 [cs] version: 2. [Online]. Available: http://arxiv.org/abs/1603.03404

[70] P. Zijlstra, "Add static_call() [LWN.net]." [Online]. Available: https://lwn.net/Articles/824406/

APPENDIX A
ADDITIONAL CASE STUDIES

In this section, we demonstrate two more case studies, where using GadgetMeter, we reveal a gadget in Linux with low exploitability and a gadget in OpenSSL expoitable with DIV contention.

A. A Gadget in Linux with Low Exploitability

In function *flock_lock_inode* of kernel, Kasper [33] reports an MDS-based gadget, as presented in Listing 5. This gadget is similar to the case study shown by Kapser in their paper, traversing a cyclic linked pointer list through the macro *list_for_each_entry* and accessing contents with the pointer. Kasper considers this gadget as exploitable because if the conditional branch on line 9 performed by the macro on line 3 is mispredicted, there's no associated data structure for *it*. As *it* is controlled by the attacker, some secret data can be accessed and leaked through the MDS-based side channel.

However, GadgetMeter scored this gadget at 3.3, suggesting that the gadget can be explored with a relatively low success rate. After carefully examining the gadget, we noticed that the branch only performs a value comparison of two pointer values. The attacker can hardly optimize this attack since it doesn't depend on memory accesses or division instructions. Still, the attack can succeed at a relatively low probability due to the simplicity of leaking instructions (one load instruction only).

B. A Gadget in OpenSSL Expoitable with DIV Contention

OpenSSL is a general-purpose cryptographic library. In function *bin2bn* of OpenSSL, SpecFuzz [48] identifies a vulnerable information flow introduced by a gadget with two branch mispredictions on line 6 and line 9 and a corrupted memory access on line 10. In this function, the program first examines whether the variable *n*, calculated from user-provided input, is within the boundary of the *ret* array. Next, the function fills up the *ret* array with a for loop, where the variable *n* acts as a boundary value. On the misprediction of branches on line 6 and line 9, the *i* on line 10 will reach out of the array boundary, and a speculative boundary check bypass on store would happen.

GadgetMeter scored this gadget at 10, suggesting that it is highly exploitable. After carefully examining the gadget, we noticed that even though the branch has no memory dependence, it depends on an integer division instruction. Therefore, by performing SMT division unit contention on a sibling thread, the attacker can largely delay the branch resolution, opening up the speculation window for secret leakage. We verified this finding with real attacks. This example demonstrates that compared with the existing naive windowing

power, our enhanced windowing power can expand the attack surface by making more gadgets exploitable in practice.

APPENDIX B
GROUND TRUTH ANALYSIS OF LIBYAML

In Table VI, we demonstrate the ground truth analysis for gadgets in the LibYAML application. For each gadget, we list the scores evaluated through different methods:

- IC stands for Instruction Counting.
- CAS stands for Cycle-Accurate Simulation.
- MAD stands for Memory Access Detection.
- GMeter stands for GadgetMeter.

Since Instruction Counting and Memory Access Detection can't provide quantitative scores, we treat "exploitable" as a score of 10 and "unexploitable" as a score of 0.

For each gadget, we also list the ground truth (GT) derived from our manual inspection. Based on the data dependencies of gadgets and all the judgments and reasonings made by different methods, we classify all gadgets into Highly Exploitable (HE), Mildly Exploitable (ME), and Unexploitable (UE). An evaluation score is classified as an under/over-estimation if it's smaller/larger than the ground truth by 5 out of 10. We also provide our reasoning for the misclassifications made by different methods in the last column. Misclassifications of other tools are colored in purple while those of GadgetMeter are colored in orange. The reasons are as follows:

- **A**—false positive of Instruction Counting due to the inability to model timing conditions, as in Listings 1, 2b and 2c.
- **B**—false negative of Cycle-Accurate Simulation due to the inability to simulate windowing power, as in Listing 2a.
- **C**—false positive of Memory Access Detection due to the limited windowing strategy, which slows down all data dependencies, as in Listings 2b and 2c.
- **D**—false negative of Memory Access Detection due to the limited windowing capabilities, as in Listing 2d.
- **E**—false positive of GadgetMeter due to the conservative window measurement to include/exclude unclear dependencies.
- **F**—false negative of GadgetMeter due to the suboptimal attack pattern sourced from an inaccurate iDAG.

The ground truth analysis for other applications and the Linux kernel is presented at https://github.com/qiling07/GadgetMeter.git.

TABLE VI: Ground truth analysis of LibYAML application.

Gadget	IC	CAS	MAD	GMeter	GT	Reasoning
1	10.0	8.55	10.0	10.0	HE	/
2	10.0	5.04	10.0	10.0	HE	/
3	10.0	3.34	10.0	3.09	ME	AC
4	10.0	6.34	10.0	10.0	HE	/
5	10.0	2.18	10.0	10.0	HE	B
6	10.0	4.8	10.0	9.99	HE	B
7	10.0	6.68	10.0	5.52	ME	/
8	10.0	9.39	10.0	9.72	HE	/
9	10.0	9.34	10.0	9.87	HE	/
10	10.0	9.63	10.0	9.79	HE	/
11	10.0	9.51	10.0	9.79	HE	/
12	10.0	9.45	10.0	9.86	HE	/
13	10.0	9.5	10.0	9.84	HE	/
14	10.0	9.23	10.0	9.26	HE	/
15	10.0	9.39	10.0	9.57	HE	/
16	10.0	9.48	10.0	9.52	HE	/
17	10.0	9.37	10.0	9.57	HE	/
18	10.0	9.32	10.0	9.38	HE	/
19	10.0	9.37	10.0	9.59	HE	/
20	10.0	2.8	10.0	2.81	ME	AC
21	10.0	4.98	10.0	10.0	HE	B
22	10.0	2.86	10.0	2.98	ME	AC
23	10.0	0.72	10.0	0.56	UE	AC
24	10.0	8.74	10.0	10.0	HE	/
25	10.0	6.13	10.0	10.0	HE	/
26	10.0	5.33	10.0	10.0	HE	/
27	10.0	3.41	10.0	10.0	HE	B
28	10.0	3.24	10.0	10.0	HE	B
29	10.0	5.36	10.0	10.0	HE	/
30	10.0	3.43	10.0	10.0	HE	B
31	10.0	8.7	10.0	10.0	HE	/
32	10.0	1.83	10.0	2.12	ME	AC
33	10.0	5.82	10.0	5.54	ME	/
34	10.0	3.45	10.0	9.99	HE	B
35	10.0	7.85	10.0	10.0	HE	/
36	10.0	7.38	10.0	10.0	HE	/
37	10.0	7.91	10.0	8.59	ME	/
38	10.0	4.48	10.0	5.45	ME	/
39	10.0	9.35	10.0	9.67	HE	/
40	10.0	9.47	10.0	9.71	HE	/
41	10.0	9.71	10.0	9.89	HE	/
42	10.0	9.54	10.0	9.82	HE	/
43	10.0	9.47	10.0	9.62	HE	/
44	10.0	9.74	10.0	9.91	HE	/
45	10.0	9.29	10.0	9.42	HE	/
46	10.0	9.31	10.0	9.61	HE	/
47	10.0	9.3	10.0	9.66	HE	/
48	10.0	7.85	10.0	10.0	HE	/
49	10.0	7.81	10.0	10.0	HE	/
50	10.0	2.52	0.0	3.05	ME	A
51	10.0	1.38	10.0	9.99	HE	B
52	10.0	3.44	10.0	2.52	ME	AC
53	10.0	3.27	10.0	3.56	ME	AC
54	10.0	4.47	10.0	10.0	HE	B
55	10.0	2.21	10.0	10.0	HE	B
56	10.0	9.26	10.0	10.0	HE	/

TABLE VI (Continued): Ground truth analysis of LibYAML application.

Gadget	IC	CAS	MAD	GMeter	GT	Reasoning	Gadget	IC	CAS	MAD	GMeter	GT	Reasoning
57	10.0	9.46	10.0	10.0	HE	/	121	10.0	1.75	10.0	1.86	ME	AC
58	10.0	1.36	10.0	10.0	HE	B	122	10.0	1.72	10.0	1.75	ME	AC
59	10.0	3.33	10.0	10.0	HE	B	123	10.0	1.5	0.0	2.39	ME	A
60	10.0	9.57	10.0	9.79	HE	/	124	10.0	2.03	10.0	3.03	ME	AC
61	10.0	9.0	10.0	9.57	HE	/	125	10.0	3.03	0.0	3.84	ME	A
62	10.0	9.34	10.0	9.57	HE	/	126	10.0	2.12	10.0	10.0	HE	B
63	10.0	9.52	10.0	9.83	HE	/	127	10.0	2.02	10.0	2.48	ME	AC
64	10.0	9.5	10.0	9.75	HE	/	128	10.0	2.11	10.0	2.43	ME	AC
65	10.0	9.54	10.0	9.84	HE	/	129	10.0	2.4	0.0	2.07	ME	A
66	10.0	9.79	10.0	9.89	HE	/	130	10.0	3.78	10.0	3.48	ME	AC
67	10.0	9.71	10.0	9.83	HE	/	131	10.0	2.63	10.0	2.63	ME	AC
68	10.0	1.27	10.0	10.0	HE	B	132	10.0	2.81	0.0	3.12	ME	A
69	10.0	9.45	10.0	10.0	HE	/	133	10.0	3.27	10.0	10.0	HE	B
70	10.0	9.57	10.0	10.0	HE	/	134	10.0	3.55	10.0	3.74	ME	AC
71	10.0	1.48	10.0	10.0	HE	B	135	10.0	3.59	10.0	3.38	ME	AC
72	10.0	2.69	10.0	2.7	ME	AC	136	10.0	2.55	10.0	2.3	ME	AC
73	10.0	3.19	10.0	10.0	HE	B	137	10.0	8.9	10.0	9.47	HE	/
74	10.0	2.14	10.0	2.7	ME	AC	138	10.0	9.41	10.0	9.82	HE	/
75	10.0	6.64	10.0	10.0	HE	/	139	10.0	1.02	0.0	0.64	UE	A
76	10.0	3.24	10.0	3.5	ME	AC	140	10.0	1.7	0.0	1.03	ME	A
77	10.0	7.88	10.0	10.0	HE	/	141	10.0	1.97	0.0	1.99	ME	A
78	10.0	5.59	10.0	10.0	HE	/	142	10.0	1.45	0.0	0.05	UE	A
79	10.0	7.82	10.0	10.0	HE	/	143	10.0	1.43	0.0	1.69	ME	A
80	10.0	3.78	10.0	10.0	HE	B	144	10.0	1.04	0.0	1.7	ME	A
81	10.0	3.5	10.0	10.0	HE	B	145	10.0	1.38	0.0	1.13	ME	A
82	10.0	8.07	10.0	8.31	ME	/	146	10.0	8.02	10.0	10.0	HE	/
83	10.0	1.8	10.0	10.0	HE	B	147	10.0	1.2	0.0	1.56	ME	A
84	10.0	2.85	0.0	2.91	ME	A	148	10.0	1.18	0.0	0.75	UE	A
85	10.0	7.08	10.0	7.3	ME	/	149	10.0	4.03	10.0	10.0	HE	B
86	10.0	1.38	10.0	2.28	ME	AC	150	10.0	0.55	0.0	0.36	UE	A
87	10.0	8.08	10.0	9.99	HE	/	151	10.0	2.41	0.0	2.35	ME	A
88	10.0	8.74	10.0	10.0	HE	/	152	10.0	0.88	0.0	0.68	UE	A
89	10.0	1.42	0.0	1.4	ME	A	153	10.0	0.72	0.0	0.16	UE	A
90	10.0	3.9	10.0	10.0	HE	B	154	10.0	9.7	10.0	9.88	HE	/
91	10.0	3.71	10.0	10.0	HE	B	155	10.0	9.34	10.0	9.55	HE	/
92	10.0	3.52	10.0	10.0	HE	B	156	10.0	9.74	10.0	9.8	HE	/
93	10.0	0.87	10.0	0.34	ME	F	157	10.0	9.24	10.0	9.45	HE	/
94	10.0	1.94	0.0	1.85	ME	A	158	10.0	7.77	10.0	7.66	ME	/
95	10.0	7.06	10.0	10.0	HE	/	159	10.0	5.17	10.0	10.0	HE	/
96	10.0	3.27	10.0	3.49	ME	AC	160	10.0	7.05	10.0	10.0	HE	/
97	10.0	3.19	10.0	2.83	ME	AC	161	10.0	3.97	10.0	10.0	HE	B
98	10.0	1.75	10.0	1.82	ME	AC	162	10.0	6.33	10.0	10.0	HE	/
99	10.0	3.36	10.0	2.96	ME	AC	163	10.0	1.81	10.0	10.0	HE	B
100	10.0	3.69	10.0	10.0	HE	B	164	10.0	9.08	10.0	10.0	HE	/
101	10.0	1.83	10.0	2.36	ME	AC	165	10.0	1.22	10.0	10.0	HE	B
102	10.0	2.71	10.0	2.3	ME	AC	166	10.0	3.75	10.0	10.0	HE	B
103	10.0	1.83	10.0	2.47	ME	AC	167	10.0	9.79	10.0	9.89	HE	/
104	10.0	3.32	10.0	3.66	ME	AC	168	10.0	9.41	10.0	9.69	HE	/
105	10.0	8.55	10.0	10.0	HE	/	169	10.0	9.42	10.0	9.66	HE	/
106	10.0	1.73	10.0	2.22	ME	AC	170	10.0	9.48	10.0	9.91	HE	/
107	10.0	3.29	10.0	10.0	HE	B	171	10.0	9.43	10.0	9.38	HE	/
108	10.0	3.14	0.0	2.94	ME	A	172	10.0	9.2	10.0	9.26	HE	/
109	10.0	6.56	10.0	7.8	ME	/	173	10.0	9.35	10.0	9.68	HE	/
110	10.0	1.23	10.0	1.52	ME	AC	174	10.0	9.51	10.0	9.67	HE	/
111	10.0	1.03	10.0	10.0	ME	E	175	10.0	9.48	10.0	9.69	HE	/
112	10.0	2.98	0.0	2.85	ME	A	176	10.0	9.48	10.0	9.63	HE	/
113	10.0	1.32	10.0	0.73	UE	AC	177	10.0	2.32	10.0	10.0	HE	B
114	10.0	2.63	0.0	2.71	ME	A	178	10.0	8.81	10.0	10.0	HE	/
115	10.0	3.36	0.0	3.55	ME	A	179	10.0	6.8	10.0	6.82	ME	/
116	10.0	2.93	10.0	2.92	ME	AC	180	10.0	6.36	10.0	9.76	ME	E
117	10.0	5.43	10.0	8.2	ME	/	181	10.0	6.97	10.0	10.0	HE	/
118	10.0	1.69	0.0	2.35	ME	A	182	10.0	7.0	10.0	10.0	HE	/
119	10.0	3.7	10.0	3.87	ME	AC	183	10.0	3.1	0.0	3.18	ME	A
120	10.0	8.44	10.0	8.92	ME	/	184	10.0	8.87	10.0	10.0	HE	/