Leaking Secrets through Modern Branch Predictors in the Speculative World

Md Hafizul Islam Chowdhuryy, *Student Member, IEEE,* and Fan Yao, *Member, IEEE*

Abstract—Transient execution attacks that exploit speculation have raised significant concerns in computer systems. Typically, branch predictors are leveraged to trigger mis-speculation in transient execution attacks. In this work, we demonstrate a new class of speculation-based attacks that targets the branch prediction unit (BPU). We find that speculative resolution of conditional branches (i.e., in nested speculation) alter the states of pattern history table (PHT) in modern processors, which are not restored after the corresponding branches are later squashed. Such characteristic allows attackers to exploit the BPU as the secret transmitting medium in transient execution attacks. To evaluate the discovered vulnerability, we build a novel attack framework, *BranchSpectre*, that enables exfiltration of unintended secrets through observing speculative PHT updates (in the form of covert and side channels). We further investigate the PHT collision mechanism in the history-based predictor and the branch prediction mode transitions in Intel processors. Built upon such knowledge, we implement an ultra-high speed covert channel (*BranchSpectre-cc*) as well as two side channels (i.e., *BranchSpectre-v1* and *BranchSpectre-v2*) that merely rely on BPU for mis-speculation trigger and secret inference in the speculative domain. Notably, BranchSpectre side channels can take advantage of much simpler code patterns than those used in Spectre attacks. We present an extensive BranchSpectre code gadget analysis on a set of popular real-world application code bases followed by a demonstration of side channel attack on OpenSSL. The evaluation results show substantially wider existence and higher exploitability of BranchSpectre code patterns in real-world software. Finally, we discuss several secure branch prediction mechanisms that can mitigate transient execution attacks exploiting modern branch predictors.

Index Terms—Branch Predictor, Transient Execution Attacks, Nested Speculation, Pattern History, Side Channels.

1 Introduction

As end users are increasingly demanding higher performance from computer systems, processor vendors have been looking for every possible source of optimization and improvement in microarchitecture design. Modern processors heavily rely on speculation to offer high instruction level parallelism. Under speculation, the processor executes instructions based on certain predicted paths, which may potentially resolve to be the wrong executions (i.e., transient execution). As transient execution can defy program software semantics, the underlying speculation engine is carefully designed to ensure that executions of undesired instructions are squashed, and no *architectural state* changes are in effect for mis-speculation.

Recent advances in transient execution attacks have demonstrated the possibility of exploiting speculative execution to construct dangerous information leakage attacks. As the branch prediction unit (BPU) plays a key role in determining instructions to be fetched in the speculative path, it has been heavily exploited in these attacks to *trigger mis-speculation*. Particularly, in Spectre V1, the attacker can induce speculative access of unintended data through branch direction mistraining, which enables the later leakage of secrets through a microarchitecture side channel [1]. Although hardware-based side channels in the non-speculative domain are widely studied [2], [3], [4], [5], [6], transient execution attacks considerably empower these classical information leakage threats by expanding the attack surface to the speculative domain. While several system-level mitigation techniques are proposed [7], [8], [9], recent studies show that these techniques either are ineffective towards certain attack variants or rarely employed in userspace due to performance concerns [5], [10].

In this work, we demonstrate a new class of hardware-based information leakage that exploits BPU state updates within the speculative domain. Our key observation is that resolutions of conditional branch instruction in the *speculative path* (e.g., nested speculation) alter the states of branch pattern history—the pattern history table (PHT) in particular. More critically, these *speculative updates of PHT states* are not restored even after the squash of speculatively executed branches in modern processors. As branch instruction outcomes in the speculation domain can depend on data accessed in a domain beyond the programmer's original intention, speculative branch execution can be potentially exploited to perform transient execution attacks in which the BPU is utilized as the *secret transmitting hardware*. We systematically explore the aforementioned security vulnerability and implement a new form of BPU side/covert channel in the speculative domain, which we term **BranchSpectre**. Similar to how Spectre fuels traditional cache timing channels, BranchSpectre reveals a more severe security concern for branch predictors with the attack manifestation in the speculative world as compared to prior BPU side channels [5], [11], [12]. Furthermore, BranchSpectre exhibits two unique characteristics distinctive from existing speculation-based exploits: (i) BranchSpectre completely relies on the BPU for transient execution triggering and speculation-domain secret transmission, minimizing the hardware footprint for attackers. It can bypass the bulk of existing defense techniques mostly targeting protection in the cache hierar-

• *M. Chowdhuryy† and F. Yao§ are with the Department of Electrical and Computer Engineering, University of Central Florida, Orlando, FL. E-mail: †reyad@knights.ucf.edu, §fan.yao@ucf.edu*

chy [13], [14]; (ii) Different from Spectre attacks that depend on the relatively rare code gadget (e.g., memory access indirection [15]), BranchSpectre can utilize much simpler code patterns that are more commonly existing (e.g., branch whose conditional is based on a speculatively-loaded value). These enable even higher exploitability for BranchSpectre as the transient execution attack in real systems.

This article considerably extends our prior work [16] in the following aspects: i) We provide new insights about mode transitions and the PHT collision mechanism for hybrid branch predictor in commercial-off-the-shelf processors, which enable substantially higher efficiency in speculative secrets transmission. ii) We introduce a new variant of BranchSpectre side channel exploitation (i.e., BranchSpectre-v2) that chains arbitrary BranchSpectre gadgets through branch target poisoning, which further enhances the attack flexibility and capability over BranchSpectre-v1. iii) We conduct a comprehensive investigation of code patterns in commonly-used application binaries to quantify the existence of BranchSpectre gadgets and demonstrate a real-world BranchSpectre attack against OpenSSL on Intel processors. Our new findings further broaden the scope of the previous work and highlight the need for rethinking branch predictor designs that are secure in speculative executions. In summary, the key contributions are[1]:

- We find that speculative update of PHT states in modern processors creates a new information leakage threat that can leverage branch predictors as the transmitting medium in the speculative domain.
- We systematically explore the prediction mode transition in three recent generations of Intel processors and reverse-engineer the PHT collision mechanism under the history-based predictor. The discovery enables probing of the PHT state perturbations by an attacker with extremely high efficiency.
- We present a novel transient execution attack framework–*BranchSpectre*–that enables information leakage through BPU in various forms: *BranchSpectre-cc*, an ultra-fast covert channel that achieves up to 1.3Mbps transmission bit rate; *BranchSpectre-v1* and *BranchSpectre-v2* side channels that leverage conditional branch mistraining and branch target poisoning respectively to induce speculative PHT update in nested speculation.
- We perform extensive analysis on code bases of 10 popular open-source applications/libraries, which shows wider existence and stronger leakage capability of BranchSpectre gadgets in real systems. Further, our case study demonstrates a real-world BranchSpectre side channel on OpenSSL that achieves 97.3% bit accuracy.
- We discuss potential speculation-secure branch predictor designs that can mitigate transient execution attacks exploiting modern branch predictors.

2 Background

2.1 Branch Prediction Structure

Branch predictor is a critical per-core structure that directs the control flow of speculative execution. At a high level, the BPU is involved with two major tasks: *direction prediction* that speculatively decides whether a conditional branch is taken or not, and *destination prediction* that predicts branch target address. The BPU enables the processor to continue execution on the predicted path before the branch's outcome resolution to minimize pipeline stalls. The underlying speculation engine ensures that instructions will eventually retire in order using the re-order buffer and only committed instructions can change the architecturally visible states (e.g., architectural registers and memory).

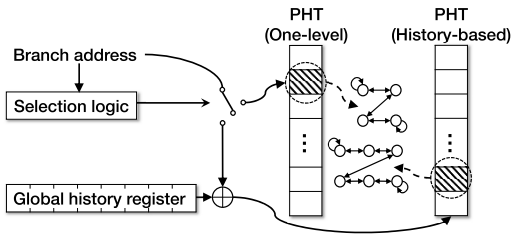

Fig. 1: A high-level illustration of modern branch predictor.

Typically, the BPU takes advantage of the Pattern History Table (PHT) for direction prediction. Each entry of the PHT incorporates a state machine using a saturating counter. For instance, in a 2-bit saturating counter, four possible states are associated with each PHT entry: *Strongly Taken* (**ST**), *Weakly Taken* (**WT**), *Weakly Not Taken* (**WN**) and *Strongly Not Taken* (**SN**). To predict branch directions, the BPU can operate in different modes [5]. Specifically, in the one-level prediction mode, the branch address is the only information source to index the PHT entry. As a result, each branch only maps to a single PHT entry. One-level prediction excels in fast training, but it performs poorly for branches whose outcomes depend on the program execution context (e.g., the execution history of recently executed branches). Differently, history-based prediction (i.e., two-level prediction) maintains the history of prior branch executions in a branch history buffer. It leverages both the branch history and branch address to access the PHT and can train multiple PHT entries, each of which corresponds to a unique branching context [16]. The history-based prediction mechanism can predict branches with complex patterns with substantially higher degree of accuracy at the expense of longer training time.

Modern processors (e.g., from Intel) generally use a hybrid design that leverages both one-level and history-based prediction in tournament mode as illustrated using *Selection logic* in Figure 1. In particular, the one-level prediction uses a predetermined number of bits from the branch address to select PHT entry. The history-based prediction instead combines the global history register (which represents the branch history state) with the branch address to index into PHT. The global history register (GHR) is a shift register that keeps track of the most recent history among all branches executed on the core. Note that since the number of entries in PHT is limited, some branches will unavoidably map to the same PHT entry, leading to PHT collision that will create interference in predictions for those branches.

2.2 Microarchitectural Timing Channel Attacks

Microarchitectural attacks are a class of information leakage threats where a malicious process manages to receive or

1. Our PoC source is released at http://tiny.cc/hfgutz.

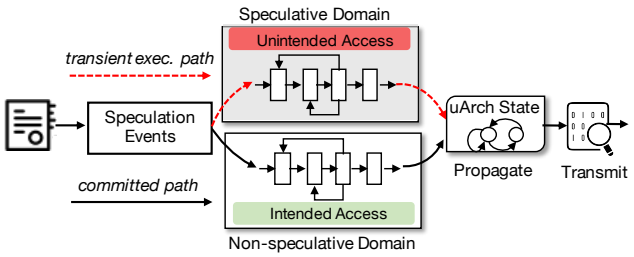

Fig. 2: Side channels in the speculation and non-speculation domain.

infer secrets via a stealthy communication channel *using microarchitectural components as the transmitting medium*. Among various attack variants, timing channels that modulate access latency to hardware resources are most widely exploited. These attacks can either manifest as covert channels that allow two isolated domains to willingly transmit data illegitimately or as side channels in which a spy process illicitly steals secrets from an unknowing victim process. Prior works have demonstrated timing channels on various hardware components in modern processors such as function units [2], caches [1], [17], and memory bus [18]. Recent works [5], [12] show that attackers can infer *program-defined secrets* that are used as branch conditionals by observing the perturbation in BPU microarchitectural states. To mitigate these classical timing channels, hardware-based techniques such as partitioning that avoid resource sharing [17], [19], [20], obfuscating timing observations [21], [22] and randomizing hardware access patterns [23] are proposed.

2.3 Transient Execution Attacks

Transient execution attacks augment classical side channels by exploiting the effect of speculation in modern processors [1]. They leverage the fact that the speculative execution path driven by the speculation engine can defy the program semantic in case of a branch misprediction. The tentative erroneous execution flow could lead to *unintended memory accesses* that cross security boundary. A successful transient execution attack depends on two factors: 1) the unintended accessed memory is propagated to taint certain microarchitectural states and 2) the microarchitectural states remain unpurged after mis-speculation is detected. Figure 2 illustrates the high-level comparison between the transient execution attack and a non-speculative side channel. While non-speculative side channels typically rely on the victim *directly* using secret-dependent control flow or data flow, transient execution attacks are even more dangerous as they substantially broaden the data outreach for an attacker.

Spectre attacks abuse branch predictors to trigger transient execution of instructions that access restricted data. These attacks widely harness caches as the target hardware component for emitting secrets as memory blocks accessed by speculative loads/stores remain in cache even after speculation is rolled back. Particularly, the V1 variant mistrains the BPU to predict the wrong branch direction, while in V2, the attacker performs branch target injecting to hijack the speculative control flow. To mitigate spectre attacks, system-level defenses are employed that aim to either limit speculation through software patches (e.g., adding fences or using retpoline [7]) or restrain branch target poisoning

```
1  if (x < bound) // Outer branch
2      // Inner branch in loop structure
3      for (int i = 0; i < iterator; ++i)
4          <some_operations>;
```

Listing 1: Sample code with potential nested speculation.

(e.g., IBRS and STIBP [8], [24]). While these techniques can mitigate security breaches due to speculation, recent studies reveal that they either do not defeat all attack vectors or may introduce non-trivial performance overhead, which hinders its adoption in userspace applications [5], [10].

3 Threat Model

Similar to previously demonstrated transient execution attacks [1], [10], we assume the attacker can run a process on the same core with the victim. These two processes are running either on the same hardware context in a round-robin fashion or on individual virtual cores under simultaneous multi-threading. Since the PHT is a per-core structure, the victim's perturbations on the PHT can potentially be observed by the attacker. The attacker process only has userspace privileges, and the pre-requisite of a malicious OS is not required. Further, we assume that the attacker has knowledge about the code/binary of the victim application and can trigger victim's execution.

4 PHT Update in Speculative Path

Speculative execution can be triggered by a multitude of on- and off-chip events with varying resolution window sizes. For instance, speculation induced by contention in functional units may be resolved within only a few cycles, while it can take thousands of cycles if triggered by a last-level cache (LLC) miss followed by a row-buffer conflict in memory [15]. As branches are one of the most common instructions in programs, multiple branch instructions may be encountered in the speculation path of a program. To avoid pipeline stall in the front-end, modern processors perform *nested speculation* where branch predictions continue to be made for branch instructions executed in the *speculative path* within a certain speculation window [25]. In case the earlier branch that triggers the speculation resolves and a misprediction is detected, the processor will squash all dependent instructions - including the subsequent branches that are fetched along the speculative path. Note that if a speculatively executed branch is resolved before it is squashed, the processor can potentially update the branch predictor's states (e.g., PHT) based on its branch outcome.

The code example from Listing 1 shows why it may be beneficial to allow speculative PHT state updates. In this example, depending on the availability of *bound* and *iterator*, the inner branch (line 3) can be resolved very quickly. In contrast, the outer branch (line 1) remains unresolved for several iterations of the inner branch. In case the outer branch is predicted as *not taken* in the direction breaking out the loop, the inner branch may be executed for multiple iterations. Once the outer branch is resolved and the actual direction of the branch is known, the processor's states are rolled back. If the PHT entry for the inner branch is not updated in the speculative path, the BPU would not

```
1  bool control;
2  if (i < bound) {    // Parent branch (b_p)
3      start = rdtsc();
4      if (control)    // Child branch (b_c)
5          <some_operations>;
6      end = rdtsc();
7  }
```

Listing 2: Testing PHT updates in speculative path.

be trained properly for predicting the loop behavior. For instance, the inner branch could be mostly *taken* while the initial PHT state for the inner branch is in the *not taken* state. In this case, the inner branch will continue to be mispredicted, leading to degraded performance if the entire speculation path turns to be useful. In contrast, if the PHT is allowed to update in the speculative path, the PHT entry will converge to *taken* after a few iterations of the inner branch.

Based on the above discussion, we can see that updating the PHT based on nested speculation can bring potential performance benefits. However, once the dependent branch is resolved and the validity of the entire speculation is refuted, the processor needs to decide how to deal with those speculative PHT updates. In particular, the processor may choose to restore the PHT to the states before speculation started. This can annul the impact of speculation with respect to the PHT perturbations by speculative branch executions in the wrong path. However, prior academic studies have shown that recovering speculative updates of the PHT when mis-speculation is detected brings negligible performance advantage [26]. In order to figure out the PHT update mechanism in real processors, we design a microbenchmark that monitors the PHT perturbations for branch executions within mis-speculated paths. The core code snippet is shown in Listing 2 where a child branch b_c can be speculatively executed when speculation is triggered by the parent branch b_p. The experiment performs the following steps:

❶ **Initialization:** In the first step, we execute a sequence of branch instructions with randomized outcomes [5] that forces the BPU to use the one-level prediction (See Section 5 below for more discussions). The one-level prediction utilizes branch address exclusively to index PHT, thus making it easier to control collision in order to infer the state of a particular PHT entry later.

❷ **Triggering b_p misprediction:** In this step, we first train the b_p branch with *not taken* outcomes and subsequently trigger mis-speculation with an out-of-bound `i` value. Based on the value of *control*, b_c will be *taken* (or *not taken*). The code segment in Listing 2 is executed multiple times so that the PHT state of b_c will converge to *taken* (or *not taken*) if BPU updates are not squashed for the transient executions of b_c. Note we load *control* in cache and flush *bound* out of cache to ensure b_c is resolved before b_p.

❸ **Infer the outcome of b_c:** In this step, we set an in-range value for i, and preload i and *bound* in cache. We then execute the code again with *control* value set to 1 (i.e., b_c should be *taken*) and measure the latency for executing the code block in Line 4-5.

We run this experiment on machines with three generations of Intel processors - *Skylake*, *Coffee Lake* and *Cascade Lake*. On each of the machines, we execute the aforementioned experiment 1000 times for each of the following configurations: 1) b_c in step ❷ *not taken*, and b_c in step ❸ *taken*; 2) b_c in both step ❷ and ❸ are *taken*. Note that this can be easily setup by controlling the value of *control*. In Figure 3, we show the latency distributions for executing the code block line 4-5 in step ❸. As we can clearly see, the execution time of the b_c in step ❸ is consistently shorter (i.e., correct prediction) if the outcome of branch b_c is the same as the *speculatively resolved outcome* of that branch in ❷. In contrast, when the outcome of b_c in step ❸ is different from step ❷, we observe longer execution time indicating a misprediction has occurred. These results evidently show that conditional branches resolved in transient execution do influence the branch prediction later on even after they are squashed. Based on the investigation results, we make the key observation that **conditional branches executed on speculative path changes the PHT state when the branch outcome is resolved, and these alterations are not restored regardless of whether the branch is eventually committed or not**. Moreover, such an observation is consistent across all processors we have evaluated. We note that the microarchitectural footprint in the BPU due to speculation can create a new avenue for transient execution attacks, essentially making it possible to exploit the BPU as the secret transmitting medium in the speculative domain. To the best of our knowledge, we are the first to investigate the exploitation of the PHT state changes in branch predictors for speculative branches in transient execution attacks. Note that while we mainly explore the speculation behavior of BPUs in Intel processors, our observation could also be applicable to chips from other vendors. Particularly, any processor that allows speculative updates of the PHT can be vulnerable to this exploitation.

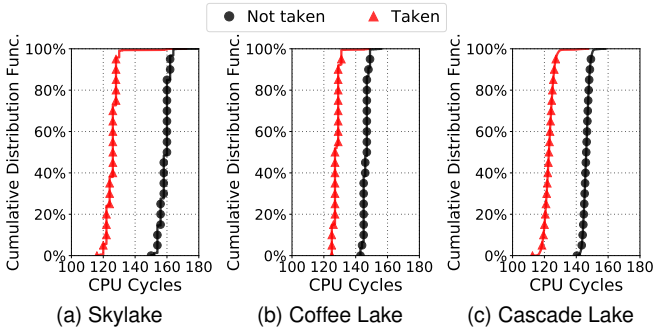

Fig. 3: Execution latency distribution for b_c branch in step ❸ corresponding to the *Taken* and *Not taken* outcome of b_c.

5 Understanding Modern Branch Prediction Mechanisms

5.1 History-based Predictor Triggering Mechanism

Understanding the branch prediction mode in operation and the conditions that trigger the BPU to transmit the prediction mode from one to another is critical for an attacker to create PHT collisions. Branchscope [5] triggers the one-level predictor mode by running a large sequence of branches (more than 100K) with random outcomes that cannot be

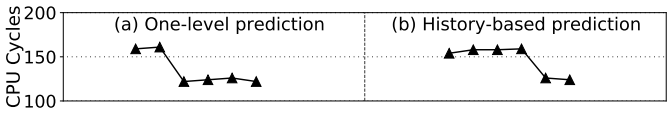

Fig. 4: Pattern of prediction performance for a consecutive execution of target branch with *Not Taken* outcome under different prediction mode. The PHT entry is initialized to *strongly taken*. Longer latency indicates misprediction.

well predicted by the history-based predictor. While one-level prediction simplifies the procedure of PHT collision, it incurs substantial runtime overhead for the attacker as the randomization procedure is needed frequently. This is because one-level prediction is typically only active for a short period before the BPU resumes the history-based prediction mode that generally exhibits better prediction accuracy. A recent study [12] has proposed an extension of the BranchScope attack by exploiting the history-based predictor. It shows that history-based activation can be enforced by running an empirically found sequence of conditional branches with a certain length. However, the exact details of the triggering mechanism have not yet been fully explored. We systematically reverse-engineer the history-based prediction transitioning mechanism, enabling efficient controls of the prediction mode in the BPU.

According to prior studies [12], the saturating counters in the PHT have distinctive sizes in different prediction modes. Particularly, 2-bit counters (4 states) are used in the one-level prediction, and 3-bit ones (8 states) are utilized in the history-based prediction. For an n-bit saturating counter, values within [0, $2^{n-1}-1$] represent *taken* states and [2^{n-1}, 2^n-1] are the *not taken* states (or vice versa). If the branch predictor is first initialized with one-level prediction, it is possible to determine whether it has transitioned to the history-based prediction based on the misprediction behavior of a certain PHT entry. Specifically, when the target PHT entry for a branch is set to *strongly taken*, executions of the same branch with *not taken* outcomes (`NNN...N`) will result in 2 mispredictions in one-level prediction but 4 mispredictions in history-based prediction before they start to predict correctly. Based on this observation, we employ a two-step procedure to determine the current prediction mode: *First*, a conditional branch is executed a sufficient number of times in one fixed direction (i.e., either *taken* or *not taken*), which trains the corresponding PHT entry to the *strong* state (i.e., all '1's or '0's in the counter). *Second*, the same branch is executed K ($K > 4$) times with the opposite branch outcome, and the execution latency of the basic block of the branch is measured. Note that while executing the same branch will guarantee the same PHT entry is accessed in one-level predictor, different PHT entries may be exercised for the same branch in history-based mode based on the GHR state. Therefore, to ensure only one PHT entry is used (in the case of history-based prediction), we run a sufficiently long sequence of predetermined branches before executing the target branch to preset the GHR. Figure 4 shows the distinctive misprediction patterns that could be used to identify whether the current prediction mode is one-level and history-based.

We hypothesize that modern processors employ a tournament-style design where the prediction accuracy of each prediction entity is dynamically monitored, and the best-performing prediction mode for each branch (or a set of branches) is selected. To evaluate the transition criteria, we create a microbenchmark that executes a target branch instruction many times with *a predetermined outcome*. This sequence can be configured such that the execution will result in a certain misprediction rate under the one-level prediction. We call such sequence the *exercising sequence*. Note that the exercising sequence for a certain misprediction rate can be generated by first initializing the PHT entry state. For example, the branch sequence `TNTNTN` will lead to 50% misprediction rate if the initial PHT state is set to `WN`. After the exercising sequence is executed, the benchmark will test the current prediction mode by executing another instruction sequence with the same target branch - the *testing sequence*. The branch outcomes during the testing sequence are set to distinguish the prediction patterns as shown in Figure 4. The overall procedure is shown in Algorithm 1.

For each exercising sequence at a specific misprediction rate (i.e., S_r), we execute the microbenchmark 100 times. The misprediction rate is confirmed through reading performance counters. At the end of each run, the program checks the current effective prediction mode. We then compute the success rate of enabling the history-based predictor. We perform the same experiment with varying lengths of

Algorithm 1: Determining the triggering of history-based prediction

Input: $t\_branch, seq\_length, mispred\_rates$
Output: $pred\_mode$

```
for r ∈ {mispred_rates} do
    // Generate outcome sequence S_r with expected
       misprediction rate r
    S_r = gen_seq (r, seq_length)
    // Execute t_branch sequence with S_r outcome
    exec (t_branch, S_r)
    // Check if history-based prediction is active
    pred_mode = chk_mode(t_branch)
Function chk_mode(t_branch):
    // Set the test outcome sequence of the target
       branch
    S_t = {TTTTTTTT, NNNN}
    exec (t_branch, S_t)
    // Check misprediction times (e.g., K set to 6)
    num_mispred = mispredictions on last K executions
    if num_mispred == 4 then
        return History-based
    else
        return One-level
```

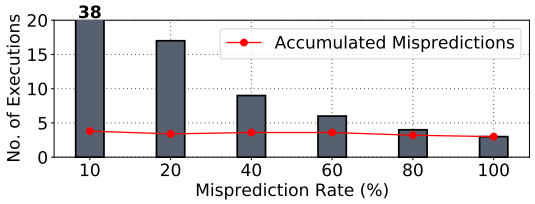

Fig. 5: Minimum number of $target\_branch$ executions required to trigger history-based prediction under different misprediction rates.

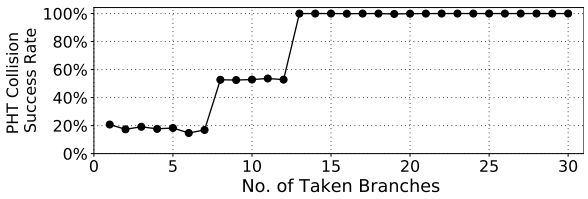

Fig. 6: Number of taken branches executed to create PHT collision under history-based predictor.

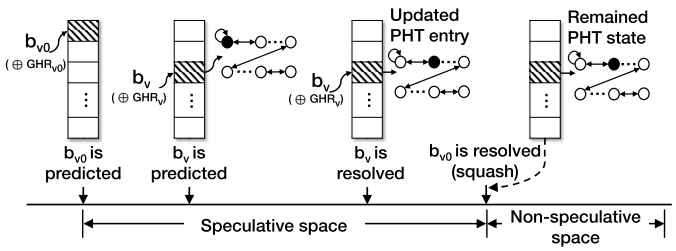

Fig. 7: Illustration of information leakage through speculative branch executions.

the *exercising sequence*. Figure 5 illustrates the minimally required length under each aimed misprediction rate. The results reveal that the required number of executions for the target branch decreases as the misprediction rate increases accordingly. More clearly, we believe that the *accumulated misprediction is used as the triggering criteria*. Particularly, when three mispredictions occur under the one-level prediction, transitioning to the history-based prediction happens. We note that such phenomenon can be potentially attributed to the internal selection logic that uses a confidence counter to choose a winning prediction mode [27]. Using this knowledge, we can swiftly trigger the history-based predictor by only executing the target branch six times with the corresponding outcomes: `TNTNTN`. Note that this sequence could either induce three or six mispredictions for one-level prediction based on the initial state of the PHT entry.

5.2 Creating PHT Collisions in History-based Predictor

With the history-based prediction, the PHT entry of a conditional branch depends on the state of branch pattern history stored in the GHR. As a result, multiple PHT entries may be trained for predicting one branch in this mode. To create a PHT collision in the history-based prediction, using a congruent branch (as in one-level prediction) is no longer sufficient. Particularly, the GHR has to be properly set for both the observee and observer branches (i.e., the branch of the victim and the attacker in side channel). One possible way to do this is to execute an excessive number of conditional branches to ensure the GHR is flushed. However, such a mechanism undermines the efficiency of PHT state inference. A more optimized approach is to precisely configure the GHR, which requires knowledge about the size of the GHR and how it is populated. Classical BPU design implements the GHR as a shift register where '1' or '0' are inserted when a conditional branch is resolved as *taken* and *not taken* respectively [28]. However, prior studies have shown that the GHR in Intel processors is populated with *partial bits* from the target addresses of *taken branches* [29]. *To determine the exact size of the GHR*, the following experiment is performed: 1) activate the history-based prediction for the target branch (as discussed in Section 5.1), 2) preset certain PHT entry to *strongly taken* by executing N distinctive $taken$ branches followed by a target branch with $taken$ outcome, 3) detect PHT collision by executing the same N number of $taken$ branches followed by the execution of target branch with $not\ taken$ outcome.

We vary N and detect PHT collision accuracy under each setting. This experiment is run 1000 times for each N value and the results are shown in Figure 6. We can see that **executing 12 taken branches before the target branch** is sufficient to preset the state of GHR for direction prediction and ensure PHT entry collision. Such observation is consistent among all processors we tested. We therefore conjecture that the size of GHR used for history-based prediction is $12 \times B_t$ where B_t is the number of bits from the targeted address of a taken branch populated to the GHR.

6 Overview of Exploitation

In this section, we show the overview of the BranchSpectre attack design which performs information leakage by inferring secrets from BPU state updates in transient execution.

As discussed in Section 5, for a PHT entry with an n-bit saturating counter with 2^n possible values, there are 1 *Strongly Taken* state, $2^{n-1}-1$ *Weakly Taken* states, $2^{n-1}-1$ *Weakly Not-taken* state and 1 *Strongly Not-taken* state. Once a PHT collision is achieved, the attacker can infer secrets by observing the sequence of prediction outcomes made for a colliding branch. Specifically, to infer the outcome of a victim branch b_v when *executed speculatively*, we first use a colliding branch b_a from the attacker's address space[2] to initialize the target PHT entry to a strong state. The attacker then triggers the execution of a victim's branch b_v speculatively. Finally, we execute b_a again to infer the state of the PHT that has already been perturbed by the victim. If the outcome of b_v is dependent on a secretive value (i.e., unintended secret), the attacker can reveal that value after the mis-speculation is corrected. Figure 7 shows a high-level implementation of the attack. Note that while we use side channel terminologies in this description, the same techniques can also be applied to covert channels. We now discuss how the attackers achieve each of these steps:

Step 1: PHT initialization for victim's branch. The attacker has two goals in this stage. First, the attacker trains the PHT entry of the victim's branch (PHT_t) so that it is pushed to a deterministic state (e.g., either *ST* or *SN*). For n-bit counters, this can be achieved by executing a branch b_a in the attacker's address space that is congruent to b_v for 2^n-1 times with the *taken* (or *not taken*) outcome. This means executions of b_a 3 and 7 times for one-level and history-based predictor respectively. Second, the attack either mistrains the branch direction prediction (in case of a conditional branch) or poisons the target address (in case of an indirect jump) of the parent instruction b_{v0} in the victim's process. This will trigger transient execution of b_{v0} in the path with the b_v

2. Note that from this point, executing a target branch in the history-based predictor will implicitly mean that GHR is properly preset to ensure collision.

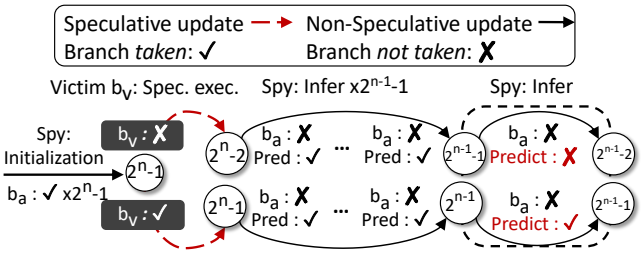

Fig. 8: Possible changes of PHT_t states under exploitation (for n-bit saturating counters). The circled values denote states of the saturating counters at each stage.

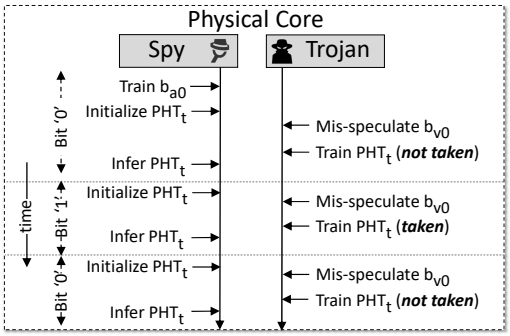

Fig. 9: Illustration of BranchSpectre covert channel protocol.

instruction. Finally, the attacker triggers the victim process execution and waits until b_v is executed speculatively.

Step 2: Victim execution in speculative path. When the victim runs, branch b_v will be first speculatively executed and later squashed. The attacker can control the speculation window for b_{v0} to make the speculation sufficiently long so that b_v is resolved first in the speculative path. b_v's speculative resolution will alter the state of PHT_t depending on certain conditionals (likely unintended data). Essentially, after victim's execution, PHT_t is tainted with the out-of-bound value.

Step 3: Infer secret by probing $\mathbf{PHT_t}$ state. In this step, the attacker aims to infer the secret value in the victim's address space by probing the state of PHT_t. To do so, the attacker executes the branch b_a that is congruent to b_v with outcome in the *opposite direction* from *Step 1*. To observe the difference, the attacker executes b_a for 2^{n-1} times and record the execution latency of b_a execution.

Figure 8 demonstrates the generalized state transition of PHT_t for n-bit counters after the attacker presets PHT_t state to *taken* in Step 1. We can see that the value of $secret$ is directly correlated with the prediction of the $2^{n-1}th$ inference operation in Step 3. Particularly, if b_v is resolved as *not taken* speculatively, the $2^{n-1}th$ branch of the attacker will be correctly predicted, otherwise, a misprediction would occur. The attacker can then infer the secret used as b_v's conditional based on timing as shown in Figure 3. Similar PHT state diagrams can be generated for branches with *not taken* outcomes in Step 1 and *taken* outcomes in Step 3.

7 BranchSpectre Covert Channel Attack

To demonstrate the information leakage threat with speculative PHT update, we investigate a covert channel where a

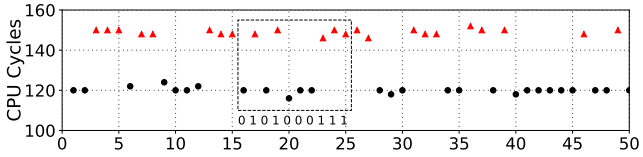

Fig. 10: Latency traces for a 50-bit transmission by *Spy* corresponding to the covert channel in Figure 9. Decoding for bit 16-25 is highlighted in the boxes.

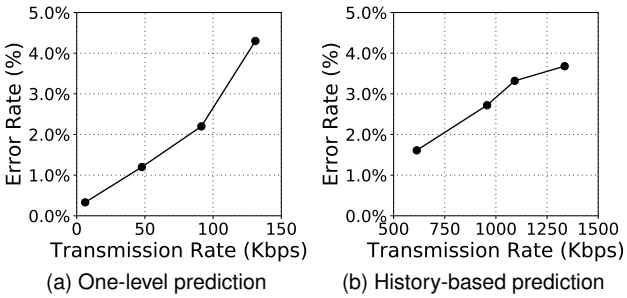

Fig. 11: Accuracy of the covert communication channel as a function of transmission rates.

trojan and a spy exploit transient branch execution to build a covert communication. We call it *BranchSpectre-cc*. Different from previous covert channel attacks in the non-speculative domain [5], [11], [12], covert channels using speculation can be more stealthy and remain undetected even with the presence of dynamic software analysis techniques [30].

To construct the BranchSpectre-cc attack, the *spy* first executes Step 1 from Section 6 and then waits for the *trojan*'s execution (Step 2) before inferring the secret in Step 3. The code gadget for trojan's exploitation can be any value-dependent conditional branch executed in the speculative path. After activating certain branch prediction mode, the attacker needs to follow this sequence of actions: *Spy initialization*–preset the PHT_t to the deterministic state by executing $b_a \rightarrow$ *Trojan training*–execute the b_v speculatively with conditional depending on sensitive data $\rightarrow$ *Spy inference*–infer the state of PHT_t after trojan's execution. Since the trojan and spy are colluding, a PHT collision between them can be achieved by simply using branches (both b_a and b_v) with the same address (for one-level prediction) or by executing the same set of 12 taken branches to preset the GHR state before the execution of b_a and b_v (for history-based prediction). Figure 9 shows the communication protocol of the covert channel and illustrates how the trojan transmits bits '010'. Figure 10 illustrates the latency traces observed by the *spy* corresponding to the 2^{n-1}th execution in the inference phase for a snippet of 50-bit transmission. The spy observes a clear pattern differentiating bit '0' and bit '1'.

There are several ways the transmission rate of the covert channel can be improved. A common optimization technique is to reuse the operation in the inference phase for one-bit transmission as initialization operation for the next bit. Specifically, we increase the number of branch executions in the inference phase to $2^n - 1$. This way, at the end of the inference, the target PHT entry is already set to a strong state (the purpose of the initialization step for

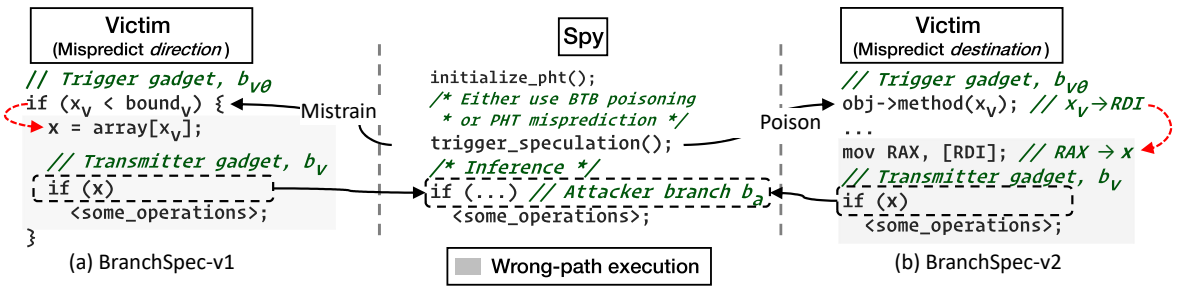

Fig. 12: High-level implementation of BranchSpectre side channels to infer *speculative changes* of PHT states. Here, (a) `x` can be unintended secret while `array` itself is not secretive in regular path of program execution or (b) `x` can be tainted by unintended memory access referenced by register `RDI`.

the next bit reception). Note that the spy infers the secret by observing the prediction from 2^{n-1}th inference branch. We can thus improve the bit rate by removing spy's *initialization* through alternating the PHT entry of b_a between ST and SN. We also perform speed-enhancing techniques particular to certain prediction mode. For instance, under one-level prediction, we can increase the bit rate by tuning bits transmitted between re-enforcements of the one-level predictor.

Figure 11 illustrates the raw transmission rate of BranchSpectre-cc and the corresponding error rate under each prediction mode. Specifically, with one-level predictor, it is observed the dominant bit rate improvement is due to coalescing multiple bits transmission for one randomization operation. The attacker can achieve a peak transmission rate of 131Kbps within 5% bit error rate (shown in Figure 11a). We find that further increasing bit rate will lead to considerable drop in bit accuracy due to transfer of the prediction mode. On the other hand, Figure 11b shows that under history-based prediction, the adversary can achieve up to 1.3Mbps with less than 4% bit error ratio, which is an order of magnitude faster than the one in one-level prediction mode. We note that BranchSpectre-cc with history-based predictor is considerably more efficient since it eliminates expensive additional operations for keeping the one-level predictor active. Compared to the existing BPU-based covert channel leveraging coarse-grained pattern history manipulations [11], BranchSpectre-cc exhibits much higher transmission rate due to precise control of collision on a single PHT entry in history-based prediction mode.

8 BranchSpectre Side Channel Attack

In this section, we demonstrate BranchSpectre side channels that enable inferring speculation-domain secrets from a victim process through branch predictor exploitation. To leverage this vulnerability, the attacker needs to find appropriate gadgets in the victim application. Particularly, BranchSpectre depends on the presence of two types of gadgets in the victim application: a *trigger gadget* to start mis-speculation and a *transmitter gadget* that perturbs a target PHT entry with the information of *speculative secret*.

Triggering gadget. Generally, any speculation inducing instruction that deviates control flow can be a trigger gadget. However, to be *exploitable*, the triggering gadget needs to fulfill two goals: (i) speculative execution in the wrong path driving towards the transmitter gadget and (ii) preparation of speculatively accessed secret (e.g., propagating the secrets to the instruction operands in the transmitter).

Transmitter gadget. The transmitter gadget can be as simple as a conditional branch that uses out-of-bound accessed data or more generally a register/memory (tainted by a secret in the speculative domain) as a part of conditional argument. Note that one such conditional jump using the tainted register is sufficient as this will alter the state of PHT for that branch according to the secret value.

Our proposed attack can manifest in ways similar to either Spectre V1 or V2, and we call our attacks *BranchSpectre-v1* and *BranchSpectre-v2* respectively. The BranchSpectre-v1 attack leverages a conditional branch to induce mis-speculation. Its speculative execution path that traverses the trigger gadget and transmitter gadget follows through the static control flow graph of the victim program. Differently, BranchSpectre-v2 harnesses branch target positioning as the triggering mechanism. The exploited path is driven by chaining the attack gadgets at potentially arbitrary locations. As a result, the speculative path in BranchSpectre-v2 is not constrained by the victim's static control flow. Since the execution path in v2-type attack can manipulate instruction sequences throughout the entire address space, it provides the adversaries with higher attack flexibility as well as code gadget availability.

8.1 Side Channel Implementation

Using the attack methodology discussed in Section 6, we can now build the two variants of BranchSpectre side channels. Specifically, the adversary locates a code sequence that corresponds to a transient execution path covering both a trigger gadget and a transmitter gadget. For BranchSpectre-v1, the code sequence contains two conditional branches where the first branch (e.g., `CMP <conditions>`→`Condtional Jump <LABEL>`) induces mis-speculation that leads to the execution of the second one with *speculative conditional values*. In BranchSpectre-v2, the trigger gadget ends with an indirect jump/call (e.g., a virtual function invocation), and its target address will be pointing to the transmitter gadget (i.e., speculative conditional branch) through branch target buffer (BTB) poisoning. Figure 12 illustrates the primary steps of side channel exploiting for BranchSpectre-v1 and BranchSpectre-v2 variants. Once the victim branch is identified and its corresponding PHT entry is determined, the attacker locates a branch in its own address space that will collide with the same PHT entry (PHT_t). As shown in Figure 12, PHT_t is first initialized by the attacker to

(a) Attacker

```
1  array1[] = [];
2
3  main() {
4   initialize_pht();
5   /* Trigger victim */
6   /* Infer */
7   start = rdtsc();
8   preset_ghr();
9   if (outer_branch) {
10   if (inner_branch) // b_a
11    <some_operations>;
12  }
13  start = rdtsc();
14 }
```

(b) Victim

```
1  sec[] = {1,1,0,1,1,
2           1,0,0,0,1};
3  array1[] = {0};
4
5  main(char *argv[]) {
6   x = atoi(argv[1]);
7   preset_ghr();
8   // Trigger gadget: b_v0
9    if (x < bound) {
10     // Transmitter gadget: b_v
11     if (array1[x])
12       sum++;
13     else sum--;
14   }
15 }
```

Listing 3: Overview of of BranchSpectre side channel PoC. `Infer` step will be executed multiple times depending on which prediction mode is used.

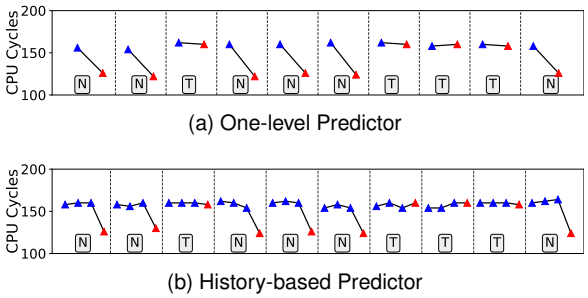

Fig. 13: Side channel attack to recover the direction of victim branch execution (shown in grayed box). Trace showing the execution latency of attacker's inference branch.

Predictor mode	Transmission rate ($Kbps$)	Accuracy (%)
One-level	0.3	99.8
History-based	188	98.4

TABLE 1: BranchSpectre side channel performance.

a predetermined state. The attacker then triggers transient execution of the victim's target branch b_v in the speculative path, which resolves before its earlier dependent branch is squashed. Lastly, the attacker infers the secrets by observing PHT_t state changes made by the speculative victim branch using the technique shown in Section 6.

Achieving PHT collision in side channels. Under the *one-level predictor*, the attacker can achieve PHT collision by using an attack branch (b_a) with the same or congruent address as the victim branch b_v. For *history-based prediction*, the attacker has to ensure that the GHR state (filled by the last 12 taken branches) before the execution of b_a is exactly the same as the one before the execution of b_v in the victim process. With a privileged attacker, this can be achieved by interrupting the victim before b_v's execution and preparing a predetermined GHR value by the attacker. However, it is potentially more challenging for unprivileged attackers since they cannot control context switch of the victim arbitrarily. Fortunately, we observe that it is not uncommon that branches leading to b_v are secret independent and they exhibit persistent prediction behaviors. As a result, it is possible for the attacker to perform off-line profiling of the victim binary with sample inputs to replicate the branch history in its inference stage. To ensure collision, gadgets exploitable for history-based prediction have the additional constraint that sufficient deterministic conditional branches preceding the transmitter gadget exist. Note that if most of the branches (not all) that impact the GHR state during the execution of the transmitter gadget can be determined, the attacker may still observe the perturbation in PHT_t by b_v by probing all possible PHT entries that would be touched.

Evaluation on PoC of BranchSpectre side channel. We first evaluate the BranchSpectre side channel on a Proof-of-concept (PoC) attack. Listing 3 shows the major sources of the attacker and victim process for BranchSpectre-v1. The victim program maintains a secretive and a non-secretive array (e.g., sec and $array1$ respectively). It also contains a trigger gadget (b_{v0}) and corresponding transmitter gadget (i.e., conditional branch, b_v). If the parent branch of b_v is mis-speculated, b_v could be executed and resolved speculatively. In the speculative domain, out-of-bound access from $array1$ could lead to load of an element from the secretive data structure sec, which alters the PHT entry state for b_v. The attacker can infer the value of one element once in sec through observing the speculative update of the targeted PHT entry. Note $preset\_ghr()$ is required only when using history-based prediction. The attacker can then exfiltrate the content of the entire array by changing the index value to $array1$. Figure 13 shows the observed latency of b_a execution for multiple iterations of the attack under one-level prediction (Figure 13a) and history-based prediction (Figure 13b). Specifically, the execution latency of b_a during the 2^{nd} inference (for 1-level predictor) or the 4^{th} inference (for history-based predictor) directly correlates with the outcome of b_v and hence, the value of the corresponding element in the sec array that was speculatively loaded. The results demonstrate that the attacker can successfully recover the exact values of the secretive array (i.e., N represents value 1 and T represents 0) *even when there is no explicit control flow or data flow dependent on it*. Table 1 lists the transmission rate and accuracy of the attack using the two branch prediction modes. Both attacks can achieve very high raw bit accuracy. Also, as expected, the side channel attack under history-based prediction can achieve much higher bit rate compared to one-level prediction (188Kbps vs. 0.3kbps) as prediction model re-enforcement is not needed in the history-based prediction mode.

Finally, it is important to note that the code gadget exploited by BranchSpectre is not vulnerable to cache-based transient execution attacks (e.g., Spectre V1). This is because there is no distinguishable data dependency on the sensitive data sec as sum is accessed in both branch directions. Additionally, even though there is a control-flow dependency on the speculative-loaded memory (if in line 11 and $else$ in line 13), the corresponding two basic blocks are mapped to the same instruction memory line, making it impossible to be inferred using cache attack such as Flush+Reload [1], [3]. In contrast, BranchSpectre is able to infer secret with this gadget by directly observing the speculative update of b_v's PHT entry, regardless of instruction memory layout and

```
1  if (x < bound) return; // b_v0
2  if (array1[x]) // b_v
3      <some_operations>;
```
(a) Consecutive conditional branches

```
1  if (x < bound) // b_v0
2      for (int i = 0; i < bound; i++)
3          if (array1[x + i]) // b_v
4              <some_operations>;
```
(b) Multi-level speculation

```
1  for (int i = x; i < bound; i++) // b_v0
2      if (array1[i])  // b_v
3          <some_operations>;
```
(c) Loop-based speculation

Listing 4: Example patterns vulnerable to BranchSpectre-v1.

data access patterns. Note that a BranchSpectre-v2 variant can be implemented similarly, with the only change of replacing the trigger gadget (b_{v0}) with an indirect jump leading to the transmitter gadget.

8.2 BranchSpectre Code Gadget Analysis

BranchSpectre exhibits several main characteristics that make it different from the classical Spectre attacks. Firstly, the proposed attack completely relies on the BPU for both accessing and inferring secrets in speculative execution path. This attack can further extend the attack surface of previously demonstrated transient execution attacks with the majority of which relying on exploiting the cache hierarchy as the secret transmitting medium [31]. Secondly, the code pattern in BranchSpectre can utilize code gadgets potentially widely existing in normal code base. For example, Spectre V1 uses *memory indirection* where an out-of-bound accessed value is used as the index to access another memory location. However, such pattern has been rarely found even in large-scale benign code bases [29]. Notably, the new attack variant exploits a simpler transmitter gadget (e.g., a branch execution based on speculation-derived conditionals), which makes it easier to find exploitable gadgets in the victim's code base.

BranchSpectre-v1 gadget in real-world applications. The key pattern of the BranchSpectre-v1 attack is nested speculation of a conditional branch where its earlier branch has been mis-speculated. Listing 4 illustrates several examples of vulnerable code patterns that can be potentially leveraged. In each of the patterns, the first branch is used to trigger the transient execution of the target branch. Note that the expression itself used as the condition of the nested branch may not be confidential in the victim's program semantic. But speculation can potentially enable access to unintended data through it (e.g., out-of-bound access). This makes BranchSpectre-v1 even more dangerous than prior branch predictor based exploitation that requires secret-dependent branching in victim's binary. We perform static program analysis to find potential BranchSpectre-v1 gadgets in real-world applications. Figure 14 shows a potential BranchSpectre-v1 in the `burl` module of lighttpd [32] web

```
static int burl_normalize_basic_unreserved_fix
   (buffer *b, buffer *t, int i, int qs) {
   /* */
    // Trigger gadget, b_v0
  for (; i < used; ++i, ++j)
     // Transmitter gadget, b_v
     if (!encoded_chars_http_uri_reqd[s[i]])
       /* */
```

Fig. 14: Potential BranchSpectre-v1 code gadget found in `burl.c` of **lighttpd**.

Binary	BranchSpectre-V2	Spectre-V2 Port contention-based
apache2	6393	1639
ld	375	138
libssl	5664	1408
nginx	6279	2185
pthread	218	64
glibc	8235	3422
libcrypto	2872	2295
lighttpd	5474	3984
openssh	7674	408
stdc	1572	720

TABLE 2: Number of BranchSpectre-v2 transmitter gadgets using memory load from any of the callee-save registers. *SS* denotes SmotherSpectre.

server. This function is used to parse the URL path of an HTTP request. In this function, the if branch in line 68 can be speculatively executed with the function argument i value that is out of the boundary of a char array $s[.]$. As a result, partial information about the data in speculative domain can be inferred through the PHT entry state that is altered. We note that an exhaustive search of BranchSpectre-v1 gadget requires comprehensive tainting and dynamic program analysis (such as symbolic execution) [33], [34], which is out of the scope of this work.

BranchSpectre-v2 gadget analysis. BranchSpectre-v2 can be more dangerous than BranchSpectre-v1 due to the possibility of chaining arbitrary code sequence. Spectre V2 implementations typically use cache side channel in the transmitter gadget. Note that BranchSpectre-v2 is considerably less stringent for the exploitability of the gadgets: *it only requires one conditional branch as the transmitter gadget*. To understand the advantage of BranchSpectre-v2 compared to other attacks in terms of gadget availability, we analyze several common real-world applications and libraries to quantify the number of exploitable gadgets. We choose SmotherSpectre [10] as the baseline, which is the prominent non-cache Spectre V2 attack that exploits port contention patterns in the transmitter gadget. Same as [10], we assume the targeted secrets are arguments of a function call in System V calling convention, which are stored in `RDI`, `RSI`, `RCX` and `RDX` in x86. As a result, a virtual function call (implemented as an indirect call) can be regarded as the trigger gadget. We perform extensive search on the transmitter gadget in common software binaries ranging from server applications to widely used C libraries. For both attacks, the transmitter gadget needs to start with a branch using a memory value pointed by any of these registers (or registers tainted by them) as the conditional. In this experiment, we only consider `TEST`→`JXX` sequence that

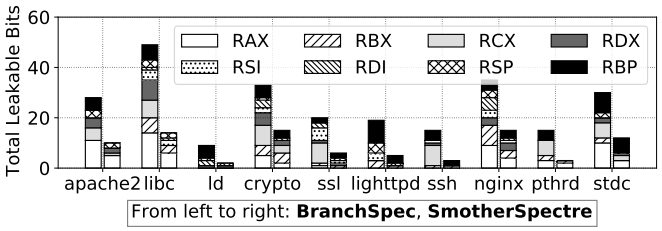

Fig. 15: Breakdown of leakable bit locations through different registers for BranchSpectre-v2 and SmotherSpectre.

```
1  if (ctx->cipher->do_cipher(ctx, out, in, inl))
2      /* */
```

(a) `do_cipher` virtual function call

```
1  0x1a8723 MOVSXD RCX, EDX // inl
2  0x1a8726 MOV RDX, QWORD PTR [RBP - 0x30] // in
3  0x1a872a MOV RSI, QWORD PTR [RBP - 0x20] // out
4  0x1a872e MOV RDI, QWORD PTR [RBP - 0x18] // ctx
5  0x1a8732 CALL RAX
```

(b) Assembly representation of Listing 5a

Listing 5: BranchSpectre-v2 triggering gadget from OpenSSL in `evp_EncryptDecryptUpdate` function.

allows leakage of information for branch conditional at the bit level. Different from BranchSpectre-v2, the transmitter gadget in SmotherSpectre (*SS*) has to exhibit distinctive port contentions between the taken and fall-through path of the target branch b_v. Table 2 shows the total number of transmitter gadgets found for BranchSpectre-v2 and SmotherSpectre. These gadgets either leak the value of the aforementioned registers or the value of memory location pointed by them. As we can see, by exploiting speculative updates in PHT, the number of code gadgets exploitable by BranchSpectre-v2 is in total 2.75 times the one for SmotherSpectre.

Since conditional branches with `TEST` typically only compares certain bits of an operand, we further investigate the locations of different bits within an 8-byte memory value that could be leaked. Figure 15 shows the number of leakable bit offsets our attack can achieve compared to SmotherSpectre via each specific register. It is observed that BranchSpectre leaks significantly more bits compared to SmotherSpectre. Overall, BranchSpectre can leak $2\times$ more bits compared to SmotherSpectre across the analyzed binaries/libraries. Note that neither attack is dependent on cache timing channel, making them resistant against most of the proposed defense mechanisms against cache timing channel.

8.3 Attack Case Study on OpenSSL

As a demonstration, we show how an unprivileged attacker can leak the secretive plaintext from a victim process running encryption using the OpenSSL. We assume the attacker process runs on the same physical core as the victim process (using SMT context). Listing 5a shows the triggering gadget used for BranchSpectre-v2 with the assembly representation showing in Listing 5b. The code gadget is within the implementation of the `EVP_EncryptDecryptUpdate` function of OpenSSL. Specifically, we target the `do_cipher`

```
1  0x1635 TEST [RDX], 0x2
2  0x1638 JNE 0x1611
```

(a) Leaking the 2nd bit

```
1  0xa24 TEST [RDX], 0x20
2  0xa2a JNE 0xC10
```

(b) Leaking the 6th bit

Listing 6: BranchSpectre-v2 transmitter gadgets in OpenSSL leaking certain bit of the first byte of secret pointed by `RDX`.

virtual function call. Note that the same vulnerability is exploited via port contention in [10]. The secret (i.e., a victim's plaintext in this case) is passed as a pointer using the 3^{rd} argument (in) to the function via `RDX` register. We observe that enough deterministic $taken$ branches can be collected by the attacker to prepare the GHR for PHT collision. The attack steps are as follows: 1) the attacker performs branch target injection [1] from its own address space to poison the indirect call in the triggering gadget to divert the control flow to the transmitter gadget. Meanwhile, the attacker also performs the target PHT initialization for history-based predictor mode, as discussed in Section 8.1. 2) The attacker triggers the encryption in the victim application, which eventually invokes `EVP_EncryptDecryptUpdate`. When the triggering gadget is encountered, the speculative control flow transfers to the transmitter gadget that alters the pre-initialized PHT entry based on the secret data. Note that here `RDX` points to the memory location of the secret in the speculative domain. 3) The attacker recovers the secret by inferring the target PHT entry similar to step 3 in Section 8.1.

Listing 6 illustrates two example transmitter gadgets found in the OpenSSL library. The conditional branch in each of the transmitter gadgets will resolve as *Taken* if the leaking bit—the second (Listing 6a) or sixth (Listing 6b) *least significant bit* of the 1^{st} secret byte—is '0'. The same branch will be resolved as *Not Taken* speculatively if the corresponding secretive bit is '1'. Note that more bits of the secret could be exfiltrated by utilizing other transmitter gadgets that leak bits at different locations. From our analysis, we have found that using the same triggering gadget, the plausible transmitter gadgets in OpenSSL can collectively leak all 8 bits of the first byte of the secret. Therefore, by chaining an additional control gadget (e.g., through ROP-like chaining [35]) that gradually shifts the offset to the address of the secret, all bytes in the secret can be leaked. We run the attack with the identified transmitter gadgets 1000 times, and our results show that the attack can successfully infer individual bits of the secrets with an average accuracy of 97.3%.

8.4 BranchSpectre in the Context of SGX

While we mainly consider a conventional threat model where attackers only have userspace accesses, BranchSpectre can be easily extended to trusted execution environments (e.g., Intel SGX) with the assumption of a privileged attacker (i.e., control over the OS). Under such scenario, the attacker can perform fine-grained scheduling of the victim's execution. We note that BranchSpectre can be more effective in the context of SGX for the following reasons: First, the high-resolution stepping (i.e., through interrupts) allows the attacker to probe the victim's speculative PHT updates *soon* after the transient execution of the victim's targeted branch,

effectively denoising the inference operation. It is worth noting that a precise interrupt right at the target branch is not possible as it is in the transient execution path. Second, the privileged attacker can replay a faulting instruction (e.g., page-fault a load [36]) before the victim's triggering gadget. This will enable BranchSpectre-v2 to integrate different transmitter gadgets over multiple rounds and potentially leak all bits of secrets in one run of the victim.

9 Discussions

9.1 Effectiveness of Existing Countermeasures

System-level defenses. The industry has rolled out several mitigation patches to limit transient execution attacks. Notably, microcode updates can prevent BTB-poisoning to higher-privilege code (IBRS [24]), eliminate indirect branch predictor sharing across SMT threads (STIBP [8]), and flush BTB upon context switches (IBPB [9]). Retpoline replaces indirect jumps with return trampoline to trap speculation [7], [37]. Similar to classical Spectre V2, BranchSpectre-v2 may be mitigated with the use of STIBP and IBPB since they restrict cross-process branch target poisoning. However, software vendors have revealed that these mechanisms essentially disable indirect branch prediction entirely [38] and are reluctant to enable them widely [39]. In fact, recent works find that such defenses have rarely been adopted in user-level applications [5], [10], leaving the *same-process* or *cross-process* userspace attack still viable. Furthermore, while avoiding secret dependent branching (as practiced in crypto algorithm implementations [40], [41]) can mitigate BPU attacks in non-speculative domain, such mechanism does not protect BranchSpectre as other irrelevant branches could be potentially exploited to access the secret speculatively and use it as the branch predicate (which defies program semantic). Finally, fencing as a general speculation protection technique can mitigate propagation of speculative secrets to the PHT. However, adding fences to all branch instructions can incur significant performance penalty [42].

Architecture-level defenses. Researchers are continuously proposing architecture-level defense mechanisms to mitigate the transient execution attacks. As the majority of these attacks primarily rely on cache to leak secret information, thwarting the cache side channel is the main focus of many architecture-level defense proposals [13], [14], [43], [44]. Since the proposed attack does not rely on the existence of cache side channel, these defenses are ineffective against BranchSpectre. Oblivious speculation frameworks prevent speculatively accessed data by restricting value propagation to microarchitecture states [45], [46], [47], [48]. These defenses ensure that microarchitecture state changes are only in place eventually for legitimate instruction executions. These techniques can defend against BranchSpectre if they are applied to audit microarchitecture states in PHT. Similar to secure cache mechanisms, randomization and isolation-based schemes for branch predictors have been proposed to eliminate cross-process BPU side channels [49], [50], [51]. These mechanisms in general limit branch predictors as the transmitter medium for information leakage, and thus can potentially mitigate the proposed attack. However, such designs typically involve considerable performance overhead and non-trivial hardware cost.

9.2 Potential Future Mitigations

We now discuss several viable mitigation techniques that can be employed in future systems.

Delaying PHT update in speculation. The PHT update for a branch instruction resolution can be delayed until the instruction is committed or can no longer be squashed by any other prior instructions. This way, the PHT will not be impacted by the branches that are resolved in the wrong path of speculation, which avoids the possibility that unintended program paths influence the state of the PHT. However, prior studies have shown early BPU updates (at resolution time) exhibit performance benefits compared to commit-time update [26]. Moreover, updates with respect to both wrong-path and correct-path branch resolutions have such positive effect. The underlying reason is that resolve-time update of a branch brings *ahead-of-time* update of PHT, which can enable fast correction of wrong prediction decisions for branch instructions in-flight. In fact, we experimentally found that branch predictors with resolve-time updates of the PHT brings on average 8% performance gain (for SPEC benchmarks) compared to resolve-time updates. Given that modern branch predictors are highly optimized hardware components, such performance difference is non-trivial. Thus, it is necessary to explore defensive strategies that retrain the performance benefit of speculatively updated processors while also ensuring information security.

PHT states obfuscation for transient branches. One straightforward mitigation against this attack is to rollback the state changes in pattern history once a mis-speculation is handled. This can be achieved by initiating checkpointing mechanisms to record the state of PHT before speculation. To do so, one checkpoint has to be created for each speculative branching point. Such mechanism can bring considerable cost of tracking the PHT state changes. This is especially the case for modern deep-pipelined processors where multi-level nested speculation is common. One potential solution to alleviate the restoration overhead is to mark the PHT entries that have been altered by transient branch instructions. However, instead of restoring those PHT entries to the original states, we could choose to obfuscate those PHT entries once speculation is terminated. Such technique is motivated by prior studies showing that perfectly checkpointing the PHT for speculative updates does not bring noticeable performance benefits [26].

Invisible PHT entry. The processor can isolate a region of PHT for the speculative branches where it records and updates the PHT state speculatively. In this scheme, when a speculatively executed branch is resolved, the shadow-PHT will be updated instead of directly updating the PHT. This shadow-PHT will be visible to other speculative branches, thus retaining the benefits of speculative update of BPU states. Each entry in the shadow-PHT can be associated with a process id, and only the branches matching the process id will be able to use this shadow-PHT entry. This will also block same-process speculative information leakage. The size of this shadow-PHT can be minimal as it only needs to keep the PHT entries corresponding to branch instructions that are in ROB and resolved speculatively. When the branch corresponding to a shadow-PHT entry is committed, the PHT entry will be merged with the original entry. Note that

conflict handling mechanism has to be implemented here since it may be possible that the same PHT entry is updated by two different processes, hence there are two copies of it in the shadow-PHT.

10 Related Work

Prior works have shown attacks exploiting BPUs to infer or transmit secrets in the *non-speculative domain* [5], [11], [12], [52]. Specifically, Evtyushkin *et al.* [11] propose a PHT-based covert channel that manipulates a large set of PHT entries to modulate timings of branch instruction executions. BranchScope [5] shows a side channel that steals intended program secrets through observing PHT state changes. BlueThunder [12] extends the exploitation of one-level prediction in [5] to two-level prediction mode with fine-grained stepping of the victim process. Furthermore, the work in [53] has revealed that changes to the BTB during speculation are persisted. With such observation, it demonstrates a spectre attack variant that uses caches as the side channel transmitter to sense BTB changes and infer secrets. We note that BranchSpectre is substantially different as none of these works has investigated the security vulnerability of *speculative PHT update* due to branch execution in speculative domain. BranchSpectre can exfiltrate unintended sensitive data by observing PHT state alterations due to branch resolutions in the transient execution path. With BranchSpectre, the transmitter gadget only needs a speculative execution of conditional branches. The end-to-end attack can be carried out by *exploiting the BPU alone* for both triggering misspeculation and transmitting speculative secrets. Our work further broadens the transient execution attack surface and motivates the need for rethinking the security of branch predictor design for speculative processors.

Transient execution attacks can be broadly categorized to *Spectre*- and *Meltdown*-type where the former relies on speculation due to prediction and the latter depends on speculation due to execution handling during retirement [31]. Previous *Spectre*-type attacks use the BPU to trigger the speculation [1], [54], while other hardware-based side channel exploits are responsible for the actual data transmission (e.g.., using caches [1] and port contentions [10]). Many research works have proposed techniques to eradicate speculation footprint on caches as countermeasures while allowing speculation to continue safely [13], [14], [43], [44]. Different from these hardware-specific defenses, oblivious speculation frameworks [44], [45], [47], [48], [55], [56] propose unified techniques for defeating transient execution attacks by tracking the propagation of speculatively accessed secret and delaying microarchitecture state changes by any instructions until they are destined to be committed. There are done at the expense of incurring various levels of performance degradation. While these generic secure speculation frameworks can be applied to mitigate BranchSpectre attacks, our preliminary studies show that the branch predictor's performance can be very sensitive to the timing of its internal state update. Therefore, there is potentially a need to design secure mechanisms that can retain the performance advantage of speculative BPU state updates.

11 Conclusion

In this work, we present BranchSpectre, an attack framework that exploits branch predictors as the transmitting medium in speculative domain in modern processors. Our key finding is that transient executions of conditional branches introduce alterations in the PHT, which will remain after these branch instructions are squashed. This allows an attacker to infer unintended secrets if speculatively executed conditional branches depend on them. Leveraging this vulnerability, we demonstrate a high bit rate speculative covert channel attack as well as V1/V2-type BranchSpectre side channel attacks. The new side channels entirely rely on exploitation of branch predictor unit and can take advantage of code patterns much simpler than the ones used in Spectre attacks. We perform code gadget analysis on several popular software code bases and demonstrate a real-world attack on OpenSSL. Our evaluation results show that BranchSpectre exhibits higher attack capability than existing works. Finally, we discuss potential mitigation mechanisms that secure branch predictors against transient execution attacks.

Acknowledgments

This material is based upon work supported in part by U.S. National Science Foundation under CNS-2008339.

References

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

[2] Z. Wang and R. B. Lee, "Covert and side channels due to processor architecture," in *IEEE ACSAC*. IEEE, 2006, pp. 473–482.

[3] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *USENIX Security*, 2014, pp. 719–732.

[4] F. Yao, M. Doroslovacki, and G. Venkataramani, "Are coherence protocol states vulnerable to information leakage?" in *IEEE HPCA*, 2018, pp. 168–179.

[5] D. Evtyushkin, R. Riley, N. Abu-Ghazaleh, and D. Ponomarev, "BranchScope: A new side-channel attack on directional branch predictor," in *ACM ASPLOS*, 2018, pp. 693–707.

[6] F. Yao, G. Venkataramani, and M. Doroslovački, "Covert timing channels exploiting non-uniform memory access based architectures," in *ACM GLSVLSI*, 2017, pp. 155–160.

[7] O. Acıiçmez, S. Gueron, and J.-P. Seifert, "Branch Target Injection CVE-2017-5715 INTEL-SA-00088," 2018.

[8] "Deep Dive: Single Thread Indirect Branch Predictors." [Online]. Available: https://software.intel.com/security-software-guidance/deep-dives/deep-dive-single-thread-indirect-branch-predictors

[9] "Deep Dive: Indirect Branch Predictor Barrier." [Online]. Available: https://software.intel.com/security-software-guidance/deep-dives/deep-dive-indirect-branch-predictor-barrier

[10] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "SMoTherSpectre: Exploiting Speculative Execution through Port Contention," in *ACM CCS*, 2019, pp. 785–800.

[11] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Understanding and mitigating covert channels through branch predictors," *ACM TACO*, vol. 13, no. 1, pp. 1–23, 2016.

[12] T. Huo, X. Meng, W. Wang, C. Hao, P. Zhao, J. Zhai, and M. Li, "Bluethunder: A 2-level directional predictor based side-channel attack against sgx," *IACR TCHES*, pp. 321–347, 2020.

[13] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "InvisiSpec: Making speculative execution invisible in the cache hierarchy," in *IEEE MICRO*, 2018, pp. 428–441.

[14] G. Saileshwar and M. K. Qureshi, "Cleanupspec: An 'undo' approach to safe speculation," in *IEEE MICRO*, 2019, pp. 73–86.

[15] A. Mambretti, M. Neugschwandtner, A. Sorniotti, E. Kirda, W. Robertson, and A. Kurmus, "Speculator: a tool to analyze speculative execution attacks and mitigations," in *ACSAC*, 2019, pp. 747–761.

[16] M. H. I. Chowdhuryy, H. Liu, and F. Yao, "BranchSpec: Information Leakage Attacks Exploiting Speculative Branch Instruction Executions," in *IEEE ICCD*, 2020.

[17] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *IEEE HPCA*, 2016, pp. 406–418.

[18] G. Venkataramani, J. Chen, and M. Doroslovacki, "Detecting hardware covert timing channels," *IEEE Micro*, vol. 36, no. 5, pp. 17–27, 2016.

[19] F. Yao, H. Fang, M. Doroslovački, and G. Venkataramani, "COTSknight: Practical defense against cache timing channel attacks using cache monitoring and partitioning technologies," in *IEEE HOST*, 2019, pp. 121–130.

[20] F. Yao, H. Fang, M. Doroslovački, and G. Venkataramani, "Leveraging cache management hardware for practical defense against cache timing channel attacks," *IEEE Micro*, vol. 39, no. 4, pp. 8–16, 2019.

[21] H. Fang, S. S. Dayapule, F. Yao, M. Doroslovački, and G. Venkataramani, "Defeating cache timing channels with hardware prefetchers," *IEEE Design & Test*, vol. 38, no. 3, pp. 7–14, 2021.

[22] F. Yao, M. Doroslovački, and G. Venkataramani, "Covert timing channels exploiting cache coherence hardware: Characterization and defense," *Springer IJPP*, vol. 47, no. 4, pp. 595–620, 2019.

[23] M. K. Qureshi, "Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *IEEE MICRO*, 2018, pp. 775–787.

[24] "Deep Dive: Indirect Branch Restricted Speculation." [Online]. Available: https://software.intel.com/security-software-guidance/insights/deep-dive-indirect-branch-restricted-speculation

[25] S. Subramanian, M. C. Jeffrey, M. Abeydeera, H. R. Lee, V. A. Ying, J. Emer, and D. Sanchez, "Fractal: An execution model for fine-grain nested speculative parallelism," in *IEEE ISCA*, 2017, pp. 587–599.

[26] E. Hao, P.-Y. Chang, and Y. N. Patt, "The effect of speculatively updating branch history on branch prediction accuracy, revisited," in *IEEE MICRO*, 1994, pp. 228–232.

[27] R. E. Kessler, "The alpha 21264 microprocessor," *IEEE Micro*, vol. 19, no. 2, pp. 24–36, 1999.

[28] S. McFarling, "Combining branch predictors," Citeseer, Tech. Rep., 1993.

[29] "Reading privileged memory with a side-channel," 2018. [Online]. Available: https://googleprojectzero.blogspot.ch/2018/01/reading-privileged-memory-with-side.html

[30] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu, "Cached: Identifying cache-based timing channels in production software," in *USENIX Security*, 2017, pp. 235–252.

[31] W. Xiong and J. Szefer, "Survey of transient execution attacks," *ACM Computing Surveys*, 2021.

[32] "Lighttpd version 1.4 (8c7dbf1)." [Online]. Available: https://github.com/lighttpd/lighttpd1.4

[33] M. C. Tol, K. Yurtseven, B. Gulmezoglu, and B. Sunar, "FastSpec: Scalable Generation and Detection of Spectre Gadgets Using Neural Embeddings," 2020.

[34] G. Wang, S. Chattopadhyay, A. K. Biswas, T. Mitra, and A. Roychoudhury, "Kleespectre: Detecting information leakage through speculative cache attacks via symbolic execution," in *ACM TOSEM*, 2020, pp. 1–31.

[35] A. Bhattacharyya, A. Sánchez, E. M. Koruyeh, N. Abu-Ghazaleh, C. Song, and M. Payer, "Specrop: Speculative exploitation of ROP chains," in *USENIX RAID)*, 2020, pp. 1–16.

[36] D. Skarlatos, M. Yan, B. Gopireddy, R. Sprabery, J. Torrellas, and C. W. Fletcher, "Microscope: Enabling microarchitectural replay attacks," in *IEEE ISCA*, 2019, pp. 318–331.

[37] P. Turner, "Retpoline: a software construct for preventing branch-target-injection," 2018. [Online]. Available: https://support.google.com/faqs/answer/7625886

[38] L. Torvalds, "[V2,27/28] x86/speculation: Add seccomp Spectre v2 user space protection mode." [Online]. Available: https://lore.kernel.org/patchwork/patch/1016829/

[39] J. Corbet, "Taming stibp." [Online]. Available: https://lwn.net/Articles/773118/

[40] A. L. Shimpi, "Mitigate a flush+reload cache attack on RSA secret exponents." [Online]. Available: https://github.com/gpg/libgcrypt/commit/e2202ff2b704623efc6277fb5256e4e15bac5676

[41] M. H. I. Chowdhuryy, R. Ewetz, A. Awad, and F. Yao, "Seeds of SEED: R-SAW: New Side Channels Exploiting Read Asymmetry in MLC Phase Change Memories," in *IEEE SEED*, 2021.

[42] M. Larabel, "The Brutal Performance Impact From Mitigating The LVI Vulnerability." [Online]. Available: https://www.phoronix.com/scan.php?page=article&item=lvi-attack-perf&num=2

[43] S. Ainsworth and T. M. Jones, "MuonTrap: Preventing Cross-Domain Spectre-Like Attacks by Capturing Speculative State," in *IEEE ISCA*, 2020, pp. 132–144.

[44] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Safespec: Banishing the spectre of a meltdown with leakage-free speculation," in *IEEE DAC*. IEEE, 2019, pp. 1–6.

[45] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, "Speculative Taint Tracking (STT) A Comprehensive Protection for Speculatively Accessed Data," in *IEEE MICRO*, 2019, pp. 954–968.

[46] J. Yu, N. Mantri, J. Torrellas, A. Morrison, and C. W. Fletcher, "Speculative data-oblivious execution: Mobilizing safe prediction for safe and efficient speculative execution," in *IEEE ISCA*, 2020, pp. 707–720.

[47] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci, "NDA: Preventing speculative execution attacks at their source," in *IEEE ISCA*, 2019, pp. 572–586.

[48] P. Li, L. Zhao, R. Hou, L. Zhang, and D. Meng, "Conditional speculation: An effective approach to safeguard out-of-order execution against spectre attacks," in *IEEE HPCA*, 2019, pp. 264–276.

[49] I. Vougioukas, N. Nikoleris, A. Sandberg, S. Diestelhorst, B. M. Al-Hashimi, and G. V. Merrett, "BRB: Mitigating Branch Predictor Side-Channels," in *IEEE HPCA*, 2019, pp. 466–477.

[50] J. Lee, Y. Ishii, and D. Sunwoo, "Securing branch predictors with two-level encryption," *ACM TACO*, vol. 17, no. 3, pp. 1–25, 2020.

[51] L. Zhao, P. Li, R. Hou, J. Li, M. C. Huang, L. Zhang, X. Qian, and D. Meng, "A lightweight isolation mechanism for secure branch predictors," *arXiv preprint arXiv:2005.08183*, 2020.

[52] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside sgx enclaves with branch shadowing," in *USENIX Security*, 2017, pp. 557–574.

[53] A. Mambretti, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, and A. Kurmus, "Two methods for exploiting speculative control flow hijacks," in *USENIX WOOT*, 2019.

[54] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *Proceedings of USENIX Security Symposium*, 2019, pp. 249–266.

[55] K. Barber, A. Bacha, L. Zhou, Y. Zhang, and R. Teodorescu, "Specshield: Shielding speculative data from microarchitectural covert channels," in *IEEE PACT*. IEEE, 2019, pp. 151–164.

[56] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander, "Efficient invisible speculative execution through selective delay and value prediction," in *IEEE ISCA*. IEEE, 2019, pp. 723–735.