BranchSpec: Information Leakage Attacks Exploiting Speculative Branch Instruction Executions

Md Hafizul Islam Chowdhuryy
University of Central Florida
Orlando, FL, USA
reyad@knights.ucf.edu

Hang Liu
Stevens Institute of Technology
Hoboken, NJ, USA
hang.liu@stevens.edu

Fan Yao
University of Central Florida
Orlando, FL, USA
fan.yao@ucf.edu

***Abstract*—Recent studies on attacks exploiting processor hardware vulnerabilities have raised significant concern for information security. Particularly, transient execution attacks such as Spectre augment microarchitectural side channels with speculative executions that lead to exfiltration of secretive data not intended to be accessed. Many prior works have demonstrated the manipulation of branch predictors for *triggering* speculative executions, and thereafter leaking sensitive information through processor microarchitectural components.**

In this paper, we present a new class of microarchitectural attack, called BranchSpec, that performs information leakage by exploiting state changes of branch predictors in *speculative path*. Our key observation is that, branch instruction executions in speculative path *alter* the states of branch pattern history, which are not restored even after the speculatively executed branches are eventually squashed. Unfortunately, this enables adversaries to harness branch predictors as the *transmitting medium* in transient execution attacks. More importantly, as compared to existing speculative attacks (e.g., Spectre), BranchSpec can take advantage of much simpler code patterns in victim's code base, making the impact of such exploitation potentially even more severe. To demonstrate this security vulnerability, we have implemented two variants of BranchSpec attacks: a side channel where a malicious spy process infers cross-boundary secrets via victim's speculatively executed nested branches, and a covert channel that communicates secrets through intentionally perturbing the branch pattern history structure via speculative branch executions. Our evaluation on Intel Skylake- and Coffee Lake-based processors reveals that these information leakage attacks are highly accurate and successful. To the best of our knowledge, this is the first work to reveal the information leakage threat due to speculative state update in branch predictor. Our studies further broaden the attack surface of processor microarchitecture, and highlight the needs for branch prediction mechanisms that are secure in transient executions.

I. Introduction

With the ever-increasing demand for computation power due to the rapid advances in software, performance optimization in hardware has continuously been the driving force in the computing industry over the past few decades. However, recent development in hardware-based attacks such as Spectre [1] have demonstrated severe vulnerabilities in modern processors that are highly optimized for performance without proper understanding of the corresponding security implications. These attacks significantly jeopardize computer system security, and have made it critical for hardware-based information security to be considered as the first-order design constraint in computer architecture.

Modern out-of-order processors heavily rely on speculation to offer high performance for software application. Under speculative execution, the processor executes a sequence of instructions tentatively, which could later be recognized to be the wrong path. The underlying speculation hardware ensures that unintended instructions are eventually squashed, and no architectural state changes relating to the wrong execution path are left. Transient execution attacks exploit the fact that speculative executions change the *microarchitectural states* of the processor, which are not cleared under incorrect speculation. For instance, in Spectre, attackers typically mistrain the Branch Prediction Unit (BPU) to trigger transient execution of instructions that access data not supposed to be reached according to program semantic (e.g., cross boundary or security domain). Such restricted data is later inferred by the attacker through inspecting the microarchitectural states of certain hardware component. Most prior works have utilized caches as the secret transmitting medium where information such as cache occupancy and coherence states are probed to exfiltrate sensitive information [2], [3]. Accordingly, many existing protection techniques for transient execution attacks target mitigating timing channels through caches [2], [4], [5].

In this work, we demonstrate a new hardware-based information leakage attack-*BranchSpec*-that exploits BPU state update *under speculative execution path*. Importantly, we observe that transient execution of conditional branches can influence the branch pattern records maintained in the Pattern History Table (PHT) of BPU. Since instructions in speculation can access data domain beyond the programmer's original intention, speculative branch execution can be potentially exploited to perform transient execution attacks where BPU is utilized as the *secret transmitting hardware*. We systematically study this vulnerability by demonstrating two variants of BranchSpec attacks: (i) a side channel where an attacker exfiltrates secrets from a victim's domain by observing the PHT state updates to a speculatively executed branch whose outcome depends on restricted data (e.g., data accessed outside of an array boundary); (ii) a high accuracy covert channel where a trojan and spy stealthily communicate by intentionally modulating the outcome of transient branch instruction executions. Our evaluation on two recent generations of processors from Intel has shown consistent observations of the discovered vulnerability, and BranchSpec is able to succeed while manifesting as both side channels and covert channels.

While information leakage attacks in hardware data structures in BPU have been demonstrated in prior works [6], [7], BranchSpec is different as it manipulates BPU state changes in *speculation domain* that enables attackers to illicitly infer data within transient execution. Essentially, BranchSpec exhibits several main characteristics that equip adversaries with stronger attack capabilities: (i) BranchSpec allows the attackers to exfiltrate unintended data through BPU in a *non-restricting* way due to speculative execution. (ii) BranchSpec completely relies on BPU for both accessing and inferring secrets in speculative execution path, therefore it can bypass the bulk of existing defensive techniques that protect the cache hierarchy [2], [4], [8], [9]. (iii) Different from classical Spectre v1 attack that depends on the rare memory access indirection code gadgets [10], BranchSpec exploits simpler code patterns (e.g., conditional branch based on an array access), which can exist more commonly in the victim's code base. Overall, our finding unveils a new facet of microarchitectural security vulnerability, which further highlights the security impact posed by transient execution attacks. In summary, the key contributions of this paper are:

- Our work identifies that conditional branch executions in speculative path that are later squashed can still alter the internal states of branch predictors in modern processors. We motivate that such vulnerabilities can potentially allow new speculation-based information leakage attacks.
- We present BranchSpec, a new attack framework that transmits/leaks secretive data through observing/modulating the state changes of branch predictors along a speculative execution path.
- We construct a side channel exploit where the spy process unrestrictedly exfiltrates secret data from victim's address space through inspecting speculative update of targeted PHT entries. We illustrate several viable code patterns that can be exploited for BranchSpec attacks. We further demonstrate an accurate covert channel using BranchSpec where the trojan intentionally transmits a speculatively accessed secret to the spy.
- We discuss potential ways to mitigate BranchSpec attacks. Our work motivates the community to perform further investigation of secure mechanisms for hardware-based information security in branch predictors.

II. Background

A. Branch Prediction Mechanism

Branch Prediction Unit (BPU) plays an influential role for the performance of speculation in modern processors. BPUs incorporate two branch prediction responsibilities: 1) predicting the outcome of a conditional branch (either to be taken or not taken), and 2) predicting the branch target address (i.e., address for the following instruction in the control flow). Typically, to predict branch outcomes, BPU takes advantage of a Pattern History Table (PHT) where a finite state machine is maintained inside each entry. Meanwhile, an internal data structure called Branch Target Buffer (BTB) keeps track of previously seen target addresses, which are used to predict future branch targets. To improve the branch prediction accuracy, modern BPUs generally feature a hybrid architecture that synergistically incorporates a basic one-level predictor [11] with a two-level *history-based predictor* [12]. The one-level predictor uses the branch address to index into the PHT where a saturating-counter is queried for branch direction prediction. While the one-level prediction mechanism can quickly adapt to changes in branch outcome, it performs poorly in the presence of complex branching patterns. In contrast, the two-level predictor takes the global history of prior executed branches together with branch addresses to make a prediction. The global history is stored in the hardware structure called Global History Register (GHR). Although history-based predictor requires longer training time, it is capable of making branch outcome prediction with high accuracy even with the presence of complex branch patterns. It has been shown that modern BPUs generally can transit between the two prediction modes to optimize prediction performance [7].

B. Side Channels and Transient Execution Attacks

Microarchitecture side channels are a form of information leakage attacks where adversaries infer secrets by observing the secret-dependent microarchitecure states left by a victim process. These attacks are extremely dangerous as they can exfiltrate sensitive information (e.g., crypto keys), bypassing many existing software-level confinement mechanism [3], [6], [13], [14]. The *transient execution* class of attacks utilizes recent advances of hardware-based side channel through the exploitation of speculative execution. Particularly, speculation allows program execution to proceed before finalizing the legitimacy of such operations. Therefore transient executions can enable update of microarchitecture states based on data *unintended to be accessed from program's perspective*. In Spectre-based attacks, the adversary often first trains the BPU to make incorrect branch outcome or target predictions, leading to speculative execution at desired program states. Secret-accessing instructions (e.g., loads) are then executed, and secret-dependent microarchitecture states are left in processor hardware components (e.g., in caches). Note that the key difference between traditional microarchitecture side channel and transient execution attack is as follows: *While the scope of leakage is limited to programmer or compiler's intended data access for the former, the latter attacks make possible the inference of data in a unrestricted way, transcending the boundary of program semantics.* Existing system-level mitigations for these attacks such as processor microcode updates and software patches (e.g., fencing) either do not defeat all attack vectors or may introduce non-trivial performance overhead. In this work, we aim to explore a new form of BPU-based information leakage attacks under transient executions.

III. Threat Model

We assume an adversarial scenario where the spy and the victim/trojan are running on the same physical core. These two processes can run either on two virtual cores when

simultaneous multi-threading (SMT) is enabled, or they share the same hardware thread in a round-robin fashion. The malicious processes (i.e., spy in side channels and spy/trojan in covert channels) are user-space processes and do not need privileged access or support from a malicious OS [7], [15]. We also assume that the attacker has knowledge about the victim program as well as the capability to trigger victim execution, which is similar to Spectre v1. We further assume that the underlying system is fully up-to-date, and system-level defense mechanisms including OS patches (e.g., Retpoline [16]) and microcode updates (e.g., IBRS [16]) are in place.

IV. PHT Update in Speculative Execution Path

Due to the existence of many events that can trigger speculative execution, the speculation window in modern processor can vary from tens of cycles to thousands of cycles [10]. As a result, a speculative execution path could encounter multiple branch instructions. Modern processor supports multiple levels of speculation, i.e., nested speculation where the processor continues to make predictions for later branch instructions and further drives the speculation path to the next level [17]. It is therefore possible that the outcome of a nested branch in the speculative path is resolved before its parent branch [10]. In this case, the processor can choose to execute along the "correct" direction[1] of the nested branch and updates the corresponding PHT entry *speculatively*.

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

Listing 1: Sample code with potential inner branch resolutions before outer branch.

To understand how PHT states update along the speculative path could be beneficial, let's consider the pseudo-code from Listing 1. In this example, depending on the availability of $bound$ and $iterator$, the outer branch may remain unresolved for several iterations of the inner branch while the inner branch can be resolved quickly. If the PHT entry for the inner branch is not allowed to update under the speculative path, the outcome for *inner branch* will not be reflected on the PHT states as long as the *outer branch* is not committed (or resolved). In that case, if the PHT state for inner branch was initially in the *not taken* state, then the BPU will continue to predict *not taken* even though the branch might be predominantly *taken* (which is typical in a *for* loop), in which case the system performance will be degraded because of considerable branch prediction errors. On the other hand, if the PHT entry is updated under speculative path then it will be adjusted to the *taken* state, and thus the branch predictor starts to have correct prediction within one or two iterations of the inner branch.

The previous discussion has shown that it is advantageous to update PHT in BPU for nested branches in speculative path before its dependent branch's resolution. However, when the dependent branch is eventually resolved and the entire speculative execution has to be squashed due to misprediction, the processor needs to determine a way to handle the PHT updates corresponding to the speculative branch instruction executions. Particularly, the BPU can choose to restore the PHT updates to the state right before the speculation starts, thus no PHT entries are polluted by the speculative branch instruction execution in the wrong path. However, two factors may hinder the adoption of such approach: 1) Restoring the initial PHT entries may require certain PHT state checkpointing mechanism that could incur higher hardware cost; 2) Even in the wrong speculative path, PHT updates due to resolution of speculative branches may still be useful under some circumstances (e.g., when the conditional values used in the speculative execution path are the same as the ones in architectural states). To find out whether PHT updates remain after speculative executions of branches that are later squashed, we develop a simple microbenchmark with core code snippet shown in Listing 2. It measures the execution time for the child branch block (b_c), which is under a parent branch b_p. The experiment performs the following steps:

```
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();
```

Listing 2: Checking impact of PHT updates in speculation.

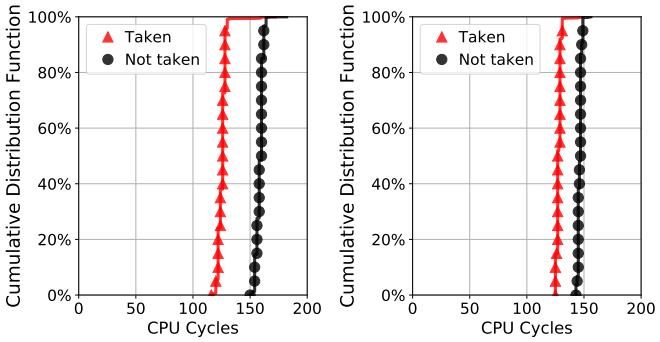
❶ **Initialization:** In the initial step, we force the BPU to use the one-level predictor by executing a sequence of branches with random outcomes. This branching pattern will decrease the accuracy of history-based predictor, and enable the use of one-level prediction mode [7]. With one-level prediction, the branch instruction address is directly used to index PHT. This makes it easier to control collision in order to infer state of a particular PHT entry later by the attacker.

❷ **Trigger speculative execution of b_c:** In this step, we first train the b_p branch with *not taken* outcomes, and trigger misspeculation by supplying a large i value. Depending on the value of $control$, b_c will either be *taken* or *not taken*. We execute the code segments twice so that the PHT state of b_c converges to either *taken* or *not taken* (if PHT updates are not squashed for transient executions of b_c). We ensure b_c is resolved before b_p by caching $control$ and flushing $bound$.

❸ **Infer the outcome of b_c:** In this step, we test whether the PHT state is altered by the execution of b_c in transient execution. Specifically, we leverage the same code in Listing 2. This time, we set i within an in-range value, and preload variable i and $bound$. 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 experiments on Intel Skylake-based and Coffee Lake-based processors 1000 times for each configuration.

[1]Note that this sub-path corresponding to the nested branch may still be squashed once its parent branch is resolved, even though this sub-path execution is based on a resolved branch outcome.



(a) Skylake Processor **(b)** Coffee Lake Processor

Fig. 1: Distribution of latency for executing the b_c code block (Line 4-5) in step ❸. The curves for *Taken* and *Not taken* correspond to the branch outcome of b_c at step ❷.

(a) Spy

```
infer(x) {
 // b_a: colliding with b_v
 if (x) <some_operations>;
}
mistrain(y) {
 // b_a0: colliding with b_v0
 if (y) <some_operations>;
}
main() {
 force_1_level_bpu();
 // Step 1: Training b_v0 and b_v
 mistrain(x_a); mistrain(x_a);
 infer(x_a); infer(x_a);
 /* Step 2: Trigger victim */
 // Step 3: Inference
 infer(!x_a);
 start = rdtsc();
 infer(!x_a); // Infer secrets
 end = rdtsc();
}
```

(b) Victim

```
main() {
 // Parent branch: b_v0
 if (x_v < bound_v) {
  /* x_v can be out of
  bound in speculation */

  x = array1[x_v];
  /* x can be unintended
  secret while array1
  itself is not secretive
  in regular path
  of program execution */

  // Victim branch: b_v
  if (x) {
    <some_operations>;
  }
}
```

Listing 3: High-level implementation of BranchSpec attacks.

Figure 1 shows the distribution of the latency samples for executing Line 4-5 in step ❸. We observe that when the outcomes of b_c in speculative path during the training phase (step ❷) is $taken$, the latency for executing the code block Line 4-5 is shorter, which indicates b_c in ❸ is correctly predicted (i.e., predicted to be *taken*). In contrast, if the speculative branch b_c is resolved to *not taken* in step ❷, b_c in step ❸ is mis-predicted (i.e., predicted *not taken*) as we can see the consistent higher execution latency. The results show that branch prediction is indeed influenced by the resolution outcome of branches in transient execution. Accordingly, we make the key observation that **conditional branch under speculative paths alters the PHT states when the outcome of the branch is resolved, regardless of whether it is committed or not**. Therefore, it is possible to leverage branch predictor as the secret transmitting source in speculation attacks. To the best of our knowledge, this is the first work that studies information leakage threat due to BPU state update for speculative branch execution.

V. BranchSpec - Overview

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

Previous work has shown that recent processors utilize a 2-bit saturating counter in each PHT entry [7]. Correspondingly, there are four possible states: *Strongly Taken* (*ST*), *Weakly Taken* (*WT*), *Weakly Not-taken* (*WN*) and *Strong Not-taken* (*SN*). Since the number of entries in the PHT structure is limited, the same PHT entry could be used by different branches (i.e., branches with colliding addresses). To unveil the state of a specific PHT entry, an attacker can observe the prediction pattern of a colliding branch in its own address space under one-level prediction. For example, if we execute an inference branch twice with *taken* outcome, and observe that the BPU is predicting *not taken* and *taken* respectively, then we can determine that the initial state of the target PHT entry is *WN*. To infer the outcome of a *speculatively executed* victim branch b_v, we can first use an attacker branch b_a with a colliding address to preset the PHT state to either *taken* or *not-taken* followed by the *speculative execution* of b_v. Then we use b_a to infer the state of PHT entry after the victim's branch execution. If b_v is executed in a speculative execution path and its outcome is dependent on a secretive value, the sensitive information could be revealed after the speculation is corrected. Listing 3 shows a high-level implementation of the BranchSpec attack. Note that while terminologies for side channels are used, this technique could also be applied to covert channels. We now discuss how the attackers achieve each of these steps:

Step 1: Prepare PHT states for victim's branches. In this step, the attacker has two goals. Firstly, the attacker trains a branch b_{a0} that is in collision with the parent branch b_{v0} in victim's process. This will trigger wrong speculation in the victim process later. Mistraining of b_{v0} can be achieved by executing b_{a0} twice with $taken$ outcome. This trains the corresponding PHT entry (PHT_p) to ST/WT irrespective of initial state of PHT_p. Secondly, it presets the state of the target PHT entry (PHT_t), which is shared by b_v and b_a. The preset operation can be done by executing b_a twice with *taken* (*not taken*), which ensures that the state of PHT_t is ST/WT (SN/WN). The attacker then invokes the execution of victim process and waits for b_v to be executed in speculation.

Step 2: Victim execution in speculative path. In this step, the victim's speculative branch execution is performed and later squashed. The state of PHT_t will be updated by b_v depending on the value of x. The attacker can evict $bound_v$ to make the speculation window for b_{v0} sufficiently long so that b_v is resolved first in the speculative path.

Step 3: Infer secret via probing $\mathbf{PHT_t}$ state. In this step, the attacker tries to infer secrets by inspecting outcome of b_v via executing b_a twice with the *opposite outcome* from *Step 1* and measure the latency of branch b_a's executions.

Figure 2 demonstrates state transition of PHT_t after each step of BranchSpec where we preset PHT_t state to *taken*

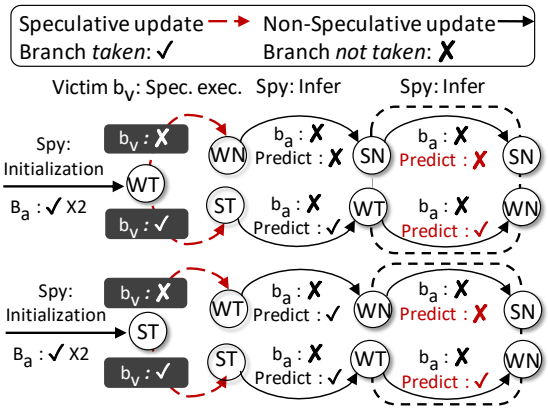


Fig. 2: Possible change of PHT_t states under exploitation.

in Step 1. We can see that the value of *secret* is directly correlated with the prediction of the second infer operation in the spy's inference stage. Similar PHT state diagrams can be generated for branches with *not taken* outcomes in Step 1 and *taken* outcomes in Step 3. Note that to easily create collisions, it is necessary to force the one-level prediction mechanism, which can be achieved as discussed in Section IV.

VI. BranchSpec: Side Channel Attacks

In this section, we demonstrate a side channel implementation of BranchSpec that infers *out-of-scope* data from a victim process via branch predictors.

A. Implementation of Side Channel

BranchSpec can be used as a side channel attack across privilege boundary as long as the attacker and victim processes run on the same physical core. Similar to Spectre, to carry out the attack, the adversary needs to first locate a vulnerable code gadget in the victim's address space. After that, the attacker runs *Step 1* as discussed in Section V before triggering the victim's execution. The rest of the steps are the same as before. Listing 4 illustrates several vulnerable code patterns commonly found in real-world applications that can be used for BranchSpec. In each of the patterns, the outer branch is used to trigger the transient execution of the inner branches. 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 BranchSpec even more dangerous than prior branch predictor based exploitation that requires secret-dependent branching in victim's binary. Listing 4 only contains a subset of vulnerable code patterns, any instruction sequence which has a nested conditional branch similar to these examples can potentially be exploited. Furthermore, prior works have shown that the Spectre V1 gadgets (i.e., memory indirection) can rarely be found in large-scale benign code base [10]. We note that to perform BranchSpec attack, the attackers essentially only have to identify code segments in victim where an array access is used to determine the outcome of a branch. Such code pattern can be more common practically.

```
1  if (x < bound) // b_v0
2     if (array1[x]) // b_v
3        <some_operations>;
```

(a) Two-level 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: BranchSpec gadgets for side channel exploitation.

B. Speculation Window and BPU-exploitation Window

For BranchSpec side channels to be successful, there are two major restrictions to overcome. One is the *speculation window*, which is the window for transient execution before the speculation is squashed. The length of speculation depends on how long it takes for the oldest branch instruction to resolve. Additionally, the maximum size of speculation window is inherently limited by the re-order buffer (ROB) size. Note that all transient execution attacks are subject to such constraint. In theory, the speculation window can be as large as the number of entries in the ROB. The attacker can manipulate the speculation window by controlling transient execution triggering events. Furthermore, as mentioned in Section V, the attack steps rely on enforcement of one-level branch prediction. However, when sufficient global branch history is collected as more branch instructions are executed, BPU will eventually transit to the two-level prediction mechanism [7]. Therefore, the number of conditional branches executed before triggering the history-based predictor is also critical for BranchSpec, which we term as *BPU-exploitation window*. Particularly, the size of BPU-exploitation window can restrict the number of conditional branches that are placed between b_{v0} and b_v for the attacker to sustain the ability to infer the outcome of b_v.

To find *BPU-exploitation window*, we conduct the following experiment. We insert consecutive n conditional branch instructions between b_{v0} and b_v, and execute the BranchSpec sequences to transmit 1000 randomly generated bits. We repeat this experiment 100 times for each n value and analyze the transmission accuracy. Since the randomly generated bits have roughly the same number of 1's and 0's, an transmission error rate close to 50% will indicate either b_v is not executed speculatively or does not directly influence the outcome of b_a.

Figure 3 shows the error rates for the side channel as the value of n changes. The window size can be determined by inspecting the point where sharp increase in error rate starts to appear. We can see that the error rate is only 0.05% when there are no additional branches between b_{v0} and b_v. Up until 39 additional branches, the bit error rates can still be within

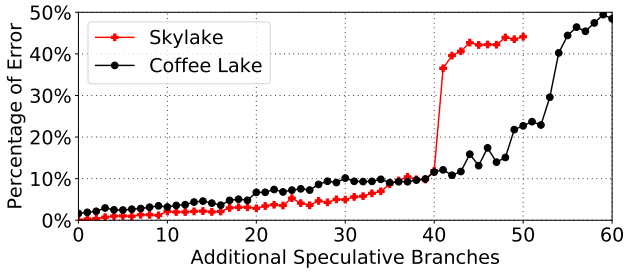


Fig. 3: Percentage of error as a function of number of additional speculatively executed branches.

10% for both the Coffee Lake- and Skylake-based systems. We note that on top of *speculation window* that limits the number of instructions to speculate (e.g., around 200 [10]), the BPU-exploitation window does not necessarily pose a more stringent constraint as branches typically mark as boundary of basic blocks containing *multiple instructions*. Finally, when the number of additional branches is more than 48 for Coffee Lake processor (40 for Skylake processor), the error rate raises high, which potentially means that at this point, the one-level predictor is no longer active.

Limiting victim's execution. Another requirement for the success of BranchSpec side channel is that the victim should not execute branch b_v more than once with different outcomes (e.g., in a loop), otherwise it may pollute the state of PHT_t relating to a specific secret. In order to precisely control execution of one single branch, previous non-speculative BPU attack [7] leveraged victim slowdown that requires fine-grained victim execution interruption (e.g., through malicious OS). Interestingly, we observe that it is possible to limit the iterations of victim branch execution by controlling the *speculation window*. To understand why this is important, let's focus on the code pattern in Listing 4(b). Here the branch b_v is inside a loop which itself is executed on the speculative path of b_{v0}. If the speculation window is large, it will allow multiple iterations of the *for* loop to be executed speculatively, then the state of PHT_t is updated by multiple executions of b_v (each based on a potentially distinct value). Under such scenario, the attacker no longer has the capability to infer the outcome of b_v corresponding to one secret reliably. However, if we limit the speculation window of b_{v0} such that only one iteration of *for* loop finishes before the b_{v0} is resolved, we can effectively control the victim branch execution without requiring explicit slowdown. Many speculation-triggering events can be manipulated to induce speculation with varying durations, including data eviction at various levels, TLB misses, different page-walk paths and page faults. Our empirical evaluation shows that the attacker can potentially control speculation window from 20 cycles to 2700 cycles, enabling attackers to have flexible control over speculation depth.

C. Implication of BranchSpec on Real-world Applications

Cryptographic algorithms. Encryption algorithms generally perform bit-wise operations. The key itself is usually stored in the address space of the encryption process. To protect against side channel attacks, cryptographic applications are recommended to not have secret-dependent branches [18]. However, these implementations may contain branches with the condition of non-secretive data [19]. Therefore, they can still be vulnerable with BranchSpec as the attacker can take control of the non-sensitive code pattern with branches to access crypto keys in the speculation domain, and thus leak secretive key bits.

Machine learning applications. With the rapid advances in machine learning (ML) techniques, ML models are increasingly regarded as the core intellectual property for model owner. In emerging ML techniques such as deep neural networks, model parameters are stored as matrices where only a small subset of parameters are non-zero. Those non-zero parameters are critical in determining model performance. Note that while the granularity of information inferred from BranchSpec can be limited in certain cases (e.g, only be able to infer if a value is zero), this attack can potentially build a shadow model with comparable performance as compared to the confidential one as ML models are typically insensitive to the magnitude of parameter values [20].

Image processing programs. Image processing application is another example which can be targeted by BranchSpec since they generally have per-pixel operations (e.g., transformation function) depending on the pixel value of an image [21]. An attacker can exploit BranchSpec code gadgets to speculatively load each pixel and leak information that could be used to recover images with high confidence.

VII. Implementation of BranchSpec Covert Channel

In this section, we discuss a covert channel implementation using BranchSpec and evaluate its effectiveness in terms of transmission accuracy and bit rates.

Construction of covert channel. To carry out a covert channel attack, the *spy* executes Step 1, then waits for the *trojan* to execute Step 2 before inferring the secret in Step 3. Note that the trojan can transmit any data from its address space due to exploitation of speculation, regardless of whether it has direct access to it or not. The trojan code gadget can be any value-dependent conditional branch executed in the speculative path. The entire sequence of execution for one bit transmission after forcing one-level BPU is as follows: *Spy Initialization:* execute b_a with *taken* outcome twice (set PHT_t to ST/WT) $\rightarrow$ *Trojan training* $\rightarrow$ *Spy Inference:* execute b_a with *not taken* outcome twice. Listing 3 can be used as a template for covert communication through modulating speculative branch executions. We implement a BranchSpec covert channel that has a high accuracy of 99.95% with the transmission rate of 6.2Kbps.

We note that the aforementioned implementation can be further optimized to improve the transmission efficiency of the covert channel attack. Firstly, as we have discussed in Section VI-B, once the one-level prediction is enabled, there

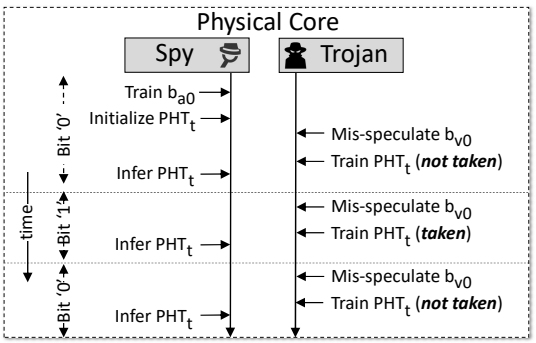
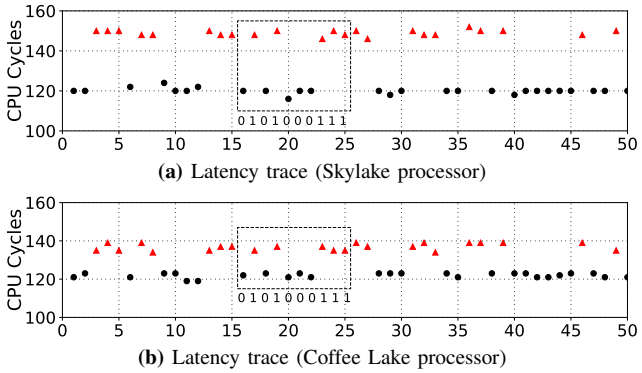


Fig. 4: Illustration of BranchSpec covert channel protocol.



(a) Latency trace (Skylake processor)

(b) Latency trace (Coffee Lake processor)

Fig. 5: Latency traces for a 50-bit transmission by *Spy* corresponding to the covert channel in Figure 4. Decoding for bit 16-25 is highlighted in the boxes.

is a BPU-exploitation window within which a number of subsequent branch predictions will be made under such mode. Therefore, instead of performing the one-level prediction enforcement operation for each bit transmission (which is time consuming), it is possible to chain the transmission of multiple bits before triggering the one-level prediction mode again. We empirically find that the attacker can chain 26 bits with a minimal transmission error rate of 3%. Secondly, the spy's initialization step can also be eliminated to simplify the transmission process of one bit. Specifically, we observe that at the end of the inference operation in *Step 3*, the targeted PHT state is always set to either ST/WT or SN/WN. If PHT_t is SN/WN, instead of executing b_v twice with *not taken* outcomes, the spy can run b_v with *taken* outcome two times and observe the execution latency for the second branch execution. We can thus remove the need for the spy's *initialization* operation by alternating the its inference operation between running b_a with *not taken* and with *taken*. The communication protocol between spy and trojan is illustrated in Figure 4. It shows how the trojan can chain 3 bits '010' with the merging of inference and initialization for consecutive bit transmissions. Figure 5 shows the latency traces (obtained by the *spy*) corresponding to the inference stage for the transmission of 50 randomly generated bits. As we can see, the spy can observe a clear pattern distinguishing bit '0' and bit '1' transmitted by the trojan for both Skylake- and Coffee Lake-based processors. Lastly, by applying these two optimizations, the BranchSpec covert channel can achieve a peak transmission rate of 131Kbps within 4% of raw error rate.

VIII. Discussion of Mitigation Techniques

The root cause of the identified vulnerability is that branch pattern history is updated by branch instructions resolved in speculative path that is later squashed. This enables a new attack vector where the adversary can leverage branch predictor for transmitting secrets in transient execution. We emphasize that BranchSpec exploits a microarchitecture-level design choice for BPU in terms of management of branch outcomes resolved in speculation. In this section we discuss several possible mitigation techniques and their tradeoffs:

Restoring BPU states for transient branches. One possible mitigation is to restore the state of BPU's pattern history to the original state after mis-speculation. This can be done by taking checkpoints for PHT state before speculation. However, such checkpointing/restoring procedure has not been applied in modern processors, potentially due to the non-trivial cost for keeping track of PHT state changes in transient execution path. Instead of accurately restoring the BPU states after mis-speculation, one potential way of alleviating the restoration overhead is to reset/obfuscate PHT states that have been altered by transient branch instructions.

Invisible PHT entry. Additionally, the processor can dedicate a segment of the PHT for speculative branches. Each PHT entry in this region will be associated with a unique process id. When a branch is resolved while being in the speculative path, instead of directly updating the corresponding PHT entry, the BPU will store the updated PHT state along with the process id separately. In case the BPU needs to make a prediction for this branch, the speculatively updated PHT will be used if the process id matches, otherwise the regular PHT entry will be utilized. When the corresponding branch is committed, the PHT entry will be merged with the original entry.

Delaying PHT update in speculation. If the PHT update for a branch instruction is delayed to the point when the instruction is committed (or at least when it can no longer be squashed by any prior instruction), branch resolutions in wrong speculation path will no longer affect BPU. This will annul the possibility that unintended program data influence the state of PHT. Note that delaying the update of BPU internal state can negatively impact the speculation performance for software applications [22].

IX. Related Work

Microarchitectural side channels have raised unprecedented concerns for information security. Transient execution attacks exploit hardware-based side channels and combine them with speculation to exfiltrate unintended data. While previous attacks mostly leveraged BPU to *trigger* speculation, our work demonstrates that BPU could be manipulated as the secret transmitting medium in transient execution attacks. Many existing transient execution attacks utilize cache timing channels

to emit secrets [5], and several mitigation techniques have been proposed to eliminate footprints of transient executions in cache hierarchy [2], [4]. These mechanisms do not defend against BranchSpec attacks. Our work reveals a new transient execution attack vector, and emphasizes the need for secure branch prediction mechanisms in speculative execution.

BranchScope [7] demonstrates a non-speculative side channel that exfiltrates data from code pattern with secret-dependent branching. Lee *et. al.* [15] have exploited the last branch record (LBR) to recover private exponents of cryptographic applications running in an enclave. BranchSpec is different from these works since it harnesses BPU state updates in the speculative path (depending on the value of speculatively accessed data). Evtyushkin *et. al.* [6] demonstrate covert channels that modulate outcomes of a large set of branches in BPU. Our BranchSpec-based covert channel has shown the use of speculative branch execution to reach high bandwidth while maintaining low error rate. Furthermore, Zhang *et.al.* [23] present a transient execution trojan that hijacks control flow by poisoning BTB in the speculative path. Since BranchSpec serves as secret-emitting speculative side channels, this technique can be applied in BranchSpec to mount attacks with even stronger capability.

There are several system-level defense techniques proposed [16], [24] to mitigate transient execution attacks. However, these techniques are ineffective against BranchSpec as they either do not defend all PHT-based attacks or could involve substantial performance overhead. Finally, several recent works have proposed architecture support to limit speculatively accessed data from propagating to instructions that can alter observable microarchitecture states [25], [26]. We envision that if deployed in real systems, these approaches can potentially be used to identify such information leakage threat.

X. Conclusion

In this paper, we present BranchSpec, a new information leakage attack in real systems that exfiltrates secrets through speculative branch executions. We observe that transient executions of conditional branches can change the PHT state in BPU, and such changes remain even if the speculatively executed branches are squashed. We find that this enables adversaries to harness branch predictor as the transmitting medium in transient execution attacks. To validate this vulnerability, we demonstrate a side channel where the attacker infers the outcome of a branch that is executed speculatively by the victim, and a covert channel that modulates speculative branch executions for illicit communication. Our evaluations show that BranchSpec is able to manifest successfully on recent Intel processors. We further discuss potential architecture-level protection techniques to mitigate the newly discovered attacks. Our work highlights the need for future research on secure branch prediction mechanisms for processor speculation.

References

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

[2] 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.

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

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

[5] W. Xiong and J. Szefer, "Survey of transient execution attacks," *arXiv preprint arXiv:2005.13435*, 2020.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] J. E. Smith, "A study of branch prediction strategies," in *ACM ISCA*, 1998, pp. 202–215.

[12] T.-Y. Yeh and Y. N. Patt, "Two-level adaptive training branch prediction," in *IEEE MICRO*, 1991, pp. 51–61.

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

[14] V. R. Kommareddy, B. Zhang, F. Yao, R. Ewetz, and A. Awad, "Are crossbar memories secure? new security vulnerabilities in crossbar memories," *IEEE CAL*, vol. 18, no. 2, pp. 174–177, 2019.

[15] 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.

[16] "Branch Target Injection / CVE-2017-5715 / INTEL-SA-00088 ." [Online]. Available: https://software.intel.com/security-software-guidance/advisory-guidance/branch-target-injection

[17] 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.

[18] "Red Hat CVE-2018-9056 hw: cpu: speculative execution branch predictor side-channel attack." [Online]. Available: https://bugzilla.redhat.com/show_bug.cgi?id=1561794

[19] O. Acıiçmez, S. Gueron, and J.-P. Seifert, "New branch prediction vulnerabilities in openssl and necessary software countermeasures," in *Springer IMACC*, 2007, pp. 185–203.

[20] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6869–6898, 2017.

[21] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *IEEE S&P*, 2015, pp. 640–656.

[22] 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.

[23] T. Zhang, K. Koltermann, and D. Evtyushkin, "Exploring branch predictors for constructing transient execution trojans," in *ACM ASPLOS*, 2020, pp. 667–682.

[24] "Bounds check bypass." [Online]. Available: https://software.intel.com/security-software-guidance/software-guidance/bounds-check-bypass

[25] 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.

[26] 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.