TR: Leakage of Sensitive Information through Speculative Branch Executions

Fan Yao Md Hafizul Islam Chowdhuryy
Department of Electrical and Computer Engineering
University of Central Florida
fan.yao@ucf.edu reyad@Knights.ucf.edu

1 Vulnerability Overview

In this report, we explain our discovery about a processor security vulnerability that allows attacker to exfiltrate secretive information by exploiting state changes maintained in branch predictors (BPU) during transient execution. Our key finding is that, *conditional 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*. This can potentially equip the adversaries with capabilities to harness branch predictors as the *transmitting medium* in transient execution attacks, which enables cache-free information leakage that steals secrets in an un-bounded fashion via BPU. With this vulnerability, the adversary can take advantage of much simpler and more common code patterns (detailed in Section 3) in software as compared to Spectre V1, which can make such exploitation even more dangerous. As far as we know, this is the first report of a transient execution attack variant that exploits branch predictor. We have tested the vulnerability using three different generations of Intel processors, and our evaluations show that such threat exists in all the systems tested. Together with this report, *we also provides two variants of workable proof-of-concept code demonstrating the new attack.* Our finding further broadens the scope of the existing Spectre attack family, and highlights the needs for new branch prediction mechanisms that are secure in speculative executions.

2 Background

Branch Prediction Unit. Modern out-of-order processors heavily rely on speculation to offer high instruction level parallelism (ILP). Branch Prediction Unit (BPU) plays an influential role for the performance of speculation–it supplies the predicted outcomes of conditional branches as well as the target addresses of indirect branches to avoid pipeline stall due to unresolved branches. Typically, BPU takes advantage of the Pattern History Table (PHT) where each entry sustains a state machine using saturating counters to predict the targeted branch outcome. For instance, with a 2-bit saturating counter, there are four possible states associated with each PHT entry: *Strongly Taken*, *Weakly Taken*, *Weakly Not Taken* and *Strongly Not Taken.* To improve the branch prediction accuracy, modern BPUs generally feature a hybrid design architecture that integrates a 1-level predictor and a history-based predictor [1] as illustrated in Figure 1. The 1-level predictor uses the branch address to index into the

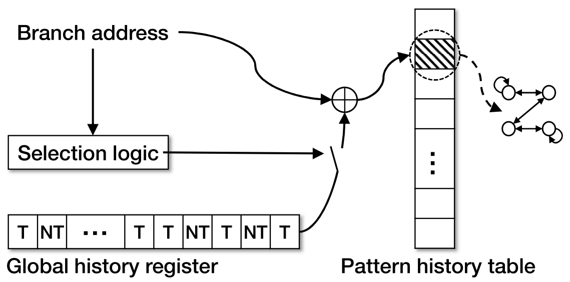

Figure 1: An illustration of the potential hybrid branch predictor architecture.

PHT 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 history-based predictor takes the global history of previously executed branches, along with branch addresses to make a prediction. The global history is stored in the hardware structure called Global History Register (GHR).

Transient Execution Attacks. Under speculative execution, the processor executes a series of instructions tentatively, which later may be recognized to be mis-speculated. Since speculative execution may *defy application semantics*, the underlying speculation hardware ensures that unintended instructions are eventually squashed, and no architectural state changes relating the wrong path are remained. Transient execution attacks exploit the fact that speculative executions alter the microarchitectural states of the processor, which may not be cleared after the mis-speculation is corrected [2–4]. Particularly, in Spectre, the attackers typically mistrain or poison the internal structure of the BPU (e.g., branch history and pattern history) to trigger transient execution of instructions that access restricted data. Such secret is later inferred by the attacker through inspecting the microarchitectural states of certain hardware components such as caches and function units [5–7].

3 Vulnerability Description

In our recent research work, we have found a potentially unknown security vulnerability in Intel processors that can enable a new variant of transient execution attacks by ***exploiting branch instruction executions in speculative path***. Our key finding is that the execution of conditional branch instructions in speculation path (i.e., nested speculative branch) alters the state of branch pattern history when they are resolved. The speculative PHT state are not restored even after the speculatively executed branches are eventually squashed (e.g., due to the mis-prediction of the parent branch). Therefore, the PHT state can reflect the values of the variable/expressions that are used as the condition of the nested branch. By carefully preparing the initial state of a targeted PHT entry and then probing the state its state change after triggering speculative execution of the victim, an attacker can potentially recover the value used by the nested branch in the victim process. Note that since the nested branch is executed in the speculative path, its conditional value can be tainted by memory that are not supposed to be accessed (i.e., crossing security boundaries). This can allow adversaries to harness branch predictors as the *transmitting medium* in *transient execution attacks*. We note that such attack vector is different from known transient execution attacks (to the best of our knowledge) where branch predictors are merely exploited to trigger mis-speculation (wrong prediction of branch outcome or address). We believe that this is the first attack that shows branch predictor could be leveraged to exfiltrate data unintended to be accessed from programmer's perspective.

```
if (x < bound) // b_p: Start of speculation
  if (array[x]) // b_v: Leaky nested branch
    some_function();
```

(a) **The new attack gadget**

```
if (x < array1_size)
  // Start of speculation
  y = array2[array1[x] * 4096];
```

(b) Spectre-style gadget

Listing 1: The new attack that leverages conditional branch in speculative execution path (a) and the original Spectre V1 exploitation (b).

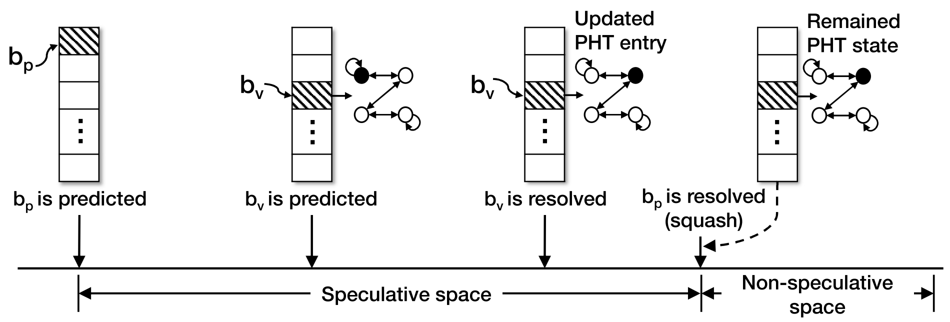

Figure 2: Illustration of the vulnerability. PHT update for b_v in speculative path remains beyond speculation.

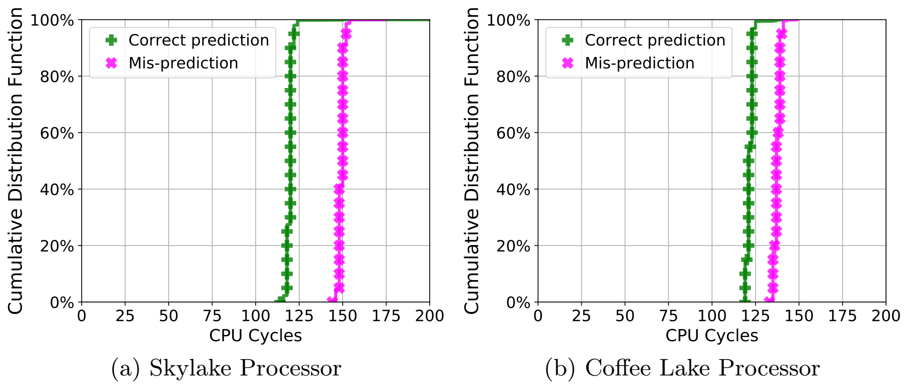

Figure 3: Distribution of latency for executing the b_v when b_v is predicted as *taken*.

We explain how the new attack variant works by using a simple code example. Listing 1a illustrates a sample code in a victim process that can be exploited by the new attack. The code snippet shows two branches in a nested fashion. Assume that x is a variable that can be controlled by the attacker, the parent branch (i.e., b_p) in Line 1 performs bound check with x, the statement shown in Line 2 then first loads the element in $array$ at index x, and the loaded value is used as the condition for the nested branch. Similar to Spectre V1, the attacker can train b_p to be taken while x is out of bound. The speculation path will then reach Line 2. Note that since x is out of bound, , $array[x]$ can point to any data in the victim address space (i.e., secrets unintended to be accessed). The exploitation of the vulnerability using the gadget shown in Listing 1 is explained with the following steps:

- **Step 1: Initialization.** In this step, the attacker trains branch b_p and b_v so that their corresponding PHT entries (saturating counters) are set to the *strongly taken* (ST) state. This could be achieved by executing branches in the attacker's address space that are congruent to b_p and b_v multiple times with the *taken* outcome. Note that the purposes for the initialization of these two branches are different. Particularly, training the PHT entry for b_p with ST is to ensure mis-speculation when the victim executes the vulnerable code. In contrast, the training of b_v is to preset its PHT entry in a deterministic state, so that secret could be exfiltrated by observing state change due to b_v's resolution in speculative path.
- **Step 2: Victim's execution.** After the branch predictor is trained by the attacker, the victim starts to execute the targeted code gadget. The variable x provided by the attacker

is out of the bound of *array*. However, since branch b_p is trained to predict taken, the processor will mistakenly choose to speculatively execute the `if` statement in Line 1 (i.e., a mis-speculation). At Line 2, the out-of-bound memory $array[x]$ is loaded and it will be used as the condition for branch b_v. Note that the attacker can intentionally prolong the time it takes to resolve b_p (e.g., by evicting the cache line holding variable *bound* out of all cache levels). In this way, the nested branch b_v can be resolved before its parent branch b_p. Upon b_v's resolution, the branch outcome is used to update the PHT entry (i.e., PHT_v). Note that *depending on the value of* $array[x]$*, the state of* PHT_v *is different.* That is, if $array[x]$ is 0, PHT_v remains in the *Strongly Taken* state, otherwise, PHT_v would be in the *Weakly Taken* (WT) state. Our experiments show that later even if b_p is resolved and the processor figures out that the speculation path is wrong, b_v's update to the PHT is not repaired. Figure 2 illustrates the victim's execution process. Essentially, after victim's execution, PHT_v is tainted with the out-of-bound value through $array[x]$.

- **Step 3: Attacker's inference.** In this step, the attacker tries to infer the value of $array[x]$ in the victim's address space by probing the state of PHT_v. To do so, the attacker can execute a test branch b_t that is congruent to b_v (i.e., these two branches collide with the same PHT entry) with known outcome (e.g., *Not Taken*). To observe the difference, the attacker execute b_t twice. The corresponding prediction outcomes for the b_t executions would be {Predict *Taken*, Predict *Taken*} and {Predict *Taken*, Predict *Not Taken*} for PHT_v in *Strongly Taken* and *Weakly Taken*, respectively. As we can see, the BPU will make a different prediction for the second execution of b_v based on the secret value. Since b_t is set to *Not Taken*, a correct prediction (Predict *Not Taken*) will result in shorter execution time for b_t branch, while a wrong prediction (Predict *Not Taken*) leads to longer execution latency. By observing the timing difference, the attacker can successfully infer the secrets in the victim's address space. Figure 3 shows the cumulative distribution function of the latency samples for execution of b_t which clearly indicates the latency difference influenced by the b_v's outcome in speculative path.

This vulnerability can be used to formulate a side-channel attack across security boundary as long as the attacker and victim processes are running on the same physical core. Similar to Spectre, to carry out the attack, the attacker first needs to locate a vulnerable code gadget in the victim's address space. Listing 2 illustrates several other vulnerable code patterns that can be used for this attack (besides the pattern in Listing 1a). In each of the patterns, the outer branch is used to trigger the transient execution of the inner branches. We assume that attacker has the control over variable x, similar to Spectre V1.

```
if (x < bound) // b_p
  for (i = 0; i < bound; i++)
    if (array[x+i]) // b_v
      <some_function>
```

(a) Multi-level branching

```
for (i = x; i < bound; i++) // b_p
  if (array[i])  // b_v
    <some_function>
```

(b) Loop-based speculation

Listing 2: Example code patterns exploitable by the new attack.

Difference from existing Spectre attacks. While bearing some similarities with Spectre V1, the new attack exhibits several main characteristics: (i) The attack completely relies on BPU for both accessing and inferring secrets in speculative execution path, therefore it can bypass the bulk of existing defense proposals against speculation attack by protecting the cache hierarchy; (ii) Different from Spectre V1 attacks that depend on the relatively rare code gadget with memory access indirection [5, 8, 9]. In fact, Google Project Zero initially reported in their blog post that they could not identify any Specter V1 vulnerable code pattern in the kernel code base [10]. Notably, the new attack variant exploits simpler code patterns (e.g., conditional branch based on an array access), which makes it easier to find exploitable code gadget in the victim's code base[1]. We believe that failure to restore PHT state after wrong speculation opens another facet for transient execution attacks.

Difference from existing BPU side channels. Recently there have been several works that demonstrate side channel attacks exploiting processor BPUs [1, 11]. Particularly, *BranchScope* [1] shows a side channel attack where an adversary leverages a novel method to create PHT collision with a victim process. The attacker then infers the victim's *program-defined secrets* by observing the branch direction, which could be correlated to the secrets. This work has been assigned with *CVE-2018-9056* in 2018 [12]. Note that while the newly discovered threat also uses PHT to exfiltrate secrets, this discovery has a key distinction: BranchScope is a *non-speculative side channel attack* on PHT while the new threat exploits PHT state update in *speculative path* due to transient branch executions that will later be squashed. The new attack can potentially make possible exfiltration of unintended data through BPU in a non-restricting way due to speculative execution, as long as the victim's nested branch is tainted with the unintended data. Differently, CVE-2018-9056 only allows exposure of the program variable (intended) used as the branch condition. In other words, this previous attack relies on the victim using secret-dependent branching (e.g., modular exponentiation algorithm). As a result, for BranchScope, system vendors such as RedHat has mentioned that "*The flaw specifically targets software which uses sensitive information in branching expressions. A software mitigation could be for the target software to avoid the use of sensitive data bits in (if..else) branching decisions...*" [13]. However, the new attack vector can not be mitigated using BranchScope's recommended fix as it can leverage branches *not* originally using secret for branching and still leak secrets due to cross trust-boundary speculative access (e.g., $array[x]$). Therefore, we envision that our discovered vulnerability has much higher severity as compared to existing BPU-based exploitations. To this end, we believe this is the first attack that shows branch predictor can be used to leak secret in out-of-bound memory location.

4 Affected Products (Tested)

We have implemented two PoC attacks. The first PoC leverages the code pattern in Listing 1a; The second PoC implements the code pattern similar to Listing 2b. All PoCs are tested on three Intel processor generations as listed below on Linux-based machines. The detailed configurations of systems as well as compilation tools are shown in the documentation of the PoC. We believe that such vulnerability is not specific to a certain operation system.

- Intel(R) Core(TM) i5-9500 (Family: Coffee Lake, Microcode: 0xd6)
- Intel(R) Core(TM) i7-6700 (Family: Skylake, Microcode: 0xdc)
- Intel(R) Xeon(R) Gold 6242 (Family: Cascade Lake, Microcode: 0x5002f01)

5 Discussion of Mitigations

Effectiveness of existing countermeasures. Current state-of-the-art mitigation/defense techniques for transient execution attacks cannot guard system against this attack, as explained in the

[1]We have not exhaustively searched the new vulnerable code patterns in software code base yet. However, we do believe such code pattern is much more common as compared to Spectre V1.

following:

- We note that existing system-level countermeasures such as Retpoline, IBRS, IBPB and STIBP are ineffective against the PHT-based transient execution attack. This is because all these mechanisms are designed to defeat transient execution attacks leveraging *poisoned branch targets*.
- Additionally, as mentioned before, rewriting program to eliminate secret dependent branching (e.g., for crypto algorithms) is not sufficient as *benign branches* (i.e., secrete-independent) in victim process can be potentially exploited to exfiltrate cross security-boundary data in speculative path. One possible software level mitigation approach is to avoid usig branches throughout the entire program. However, such implementation can incur non-trivial performance overhead and cause compatibility issues. Using fencing instructions to prevent speculation can potentially defeat Spectre attack variants, however doing this can have considerable performance degradation.

Future mitigations. We believe the root cause of the discovered vulnerability is that the PHT state update in speculative execution path is not reverted back when the speculation is deemed incorrect. We guess that the potential rationale for not performing the PHT state restoration is two-fold: (i) restoration of the PHT state after mis-speculation requires additional hardware structures that may increase the cost and complexity of BPU design. (ii) There is no strong incentive of restoring wrong PHT update due to mis-speculation as such strategy will not have noticeable impact of system performance, as studied by the research work in [14]. This vulnerability is not due to flaws in software and OS kernel implementation, and the root cause stems from the microarchitecture design of PHT. We believe the treatment of BPU-based transient execution attacks should be different from the non-speculative BPU side channels (such as BranchScope) since modern processors already have support for state restoration in hardware components for speculation. One potential protection approach for this new vulnerability is to use lightweight PHT update erasing mechanism in BPU that obfuscates the PHT state changes in transient execution. Another possible way to mitigate this vulnerability is to delay update of PHT from branch resolution time to the point when it is certain that the nested branch will eventually commit.

6 Disclosure

We have responsibly disclosed our findings in this work to Intel in September 2020.

References

[1] CVE-2018-9056. Available from MITRE, CVE-ID CVE-2018-9056., January 1 2021. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-9056.

[2] CVE-2017-5715. Available from MITRE, CVE-ID CVE-2017-5715., February 1 2017. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5715.

[3] CVE-2017-5753. Available from MITRE, CVE-ID CVE-2017-5753., February 1 2017. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5753.

[4] CVE-2017-5754. Available from MITRE, CVE-ID CVE-2017-5754., February 1 2017. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5754.

[5] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[6] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with {TLB} attacks. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 955–972, 2018.

[7] Julian Stecklina and Thomas Prescher. Lazyfp: Leaking fpu register state using microarchitectural side-channels. *arXiv preprint arXiv:1806.07480*, 2018.

[8] Andrea Mambretti, Matthias Neugschwandtner, Alessandro Sorniotti, Engin Kirda, William Robertson, and Anil Kurmus. Let's not speculate: Discovering and analyzing speculative execution attacks. *IBM Research*, 2018.

[9] Andrea Mambretti, Matthias Neugschwandtner, Alessandro Sorniotti, Engin Kirda, William Robertson, and Anil Kurmus. Speculator: a tool to analyze speculative execution attacks and mitigations. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 747–761, 2019.

[10] Reading privileged memory with a side-channel, 2018.

[11] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside {SGX} enclaves with branch shadowing. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 557–574, 2017.

[12] CVE-2018-9056. Available from MITRE, CVE-ID CVE-2018-9056., March 27 2018. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-9056.

[13] Red Hat. 1561794 - (cve-2018-9056) cve-2018-9056 hw: cpu: speculative execution branch predictor side-channel attack, 2018. Available: https://bugzilla.redhat.com/show_bug.cgi?id=1561794.

[14] Eric Hao, Po-Yung Chang, and Yale N Patt. The effect of speculatively updating branch history on branch prediction accuracy, revisited. In *Proceedings of the 27th annual international symposium on Microarchitecture*, pages 228–232, 1994.