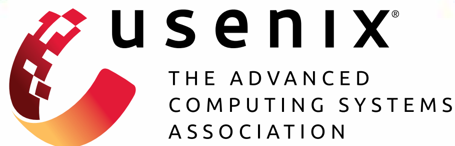

BunnyHop: Exploiting the Instruction Prefetcher

Zhiyuan Zhang, Mingtian Tao, and Sioli O'Connell, *The University of Adelaide;* Chitchanok Chuengsatiansup, *The University of Melbourne;* Daniel Genkin, *Georgia Tech;* Yuval Yarom, *The University of Adelaide*

https://www.usenix.org/conference/usenixsecurity23/presentation/zhang-zhiyuan-bunnyhop

This paper is included in the Proceedings of the
32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Proceedings of the
32nd USENIX Security Symposium
is sponsored by USENIX.

BunnyHop: Exploiting the Instruction Prefetcher

Zhiyuan Zhang[†], *Mingtian Tao*[†], *Sioli O'Connell*[†],
Chitchanok Chuengsatiansup[‡], *Daniel Genkin*[§], *Yuval Yarom*[†]

[†] *The University of Adelaide*
[‡] *The University of Melbourne*
[§] *Georgia Tech.*

Abstract

The instruction prefetcher is a microarchitectural component whose task is to bring program code into the instruction cache. To predict which code is likely to be executed, the instruction prefetcher relies on the branch predictor.

In this paper we investigate the instruction prefetcher in modern Intel processors. We first propose BunnyHop, a technique that uses the instruction prefetcher to encode branch prediction information as a cache state. We show how to use BunnyHop to perform low-noise attacks on the branch predictor. Specifically, we show how to implement attacks similar to Flush+Reload and Prime+Probe on the branch predictor instead of on the data caches. We then show that BunnyHop allows using the instruction prefetcher as a confused deputy to force cache eviction within a victim. We use this to demonstrate an attack on an implementation of AES protected with both cache coloring and data prefetch.

1 Introduction

Over the last decades, awareness of the perils of sharing microarchitectural components has constantly increased. Microarchitectural side-channel attacks [28, 71] have been shown to break multiple cryptographic implementations [6, 19, 20, 29, 50, 55, 57, 64, 74, 84] and have also affected many other applications [24, 32, 34, 65, 67, 68]. Recently, such techniques have been used in transient-execution attacks to leak information from the speculative state of the processor [14, 17, 43, 46, 73, 82, 88].

A wide range of components have been exploited for carrying out microarchitectural attacks [3, 5, 8, 23, 31, 35, 52, 56, 59, 62, 79, 83, 85]. However, there remain many components that have been investigated less, and the security implications of their operations are yet to be discovered.

One of the less explored components is the instruction prefetcher [61, 70] whose role is to cache memory blocks with the aim of reducing the future cost of executing code from these blocks. Unlike branch prediction, which can result in speculative execution of instructions, the instruction prefetcher only brings memory to the cache, but does not cause execution. Yet, at least on Intel processors, the instruction prefetcher relies on the branch prediction unit for determining memory blocks to prefetch [41].

In this work, we ask the following question:

What effects do the instruction prefetcher and branch predictor have on each other and what are the security implications of these effects?

Our Contribution

In this paper we investigate the instruction prefetcher and its implications on security.

We first reverse engineer the prefetcher on several models of Intel Core processors, finding that the processor prefetches up to 26 memory blocks, depending on the model. We analyze the interaction between branch instructions and the instruction prefetcher, showing that branch instructions train the prefetcher, whereas non-branch instructions remove the training. We further investigate the implications of hyperthreading, demonstrating that the instruction prefetcher is time-shared between the hyperthreads and that there is no cross-hyperthread training.

We then design BunnyHop, a technique that translates branch prediction state to cache state. Specifically, BunnyHop builds on the observation that when the branch predictor (more specifically, the branch target buffer (BTB)) predicts that code at a source address branches to a target address, diverting the instruction prefetcher to the source address will bring the target address to the cache.

We combine this observation with the Flush+Reload attack [33, 84] to interrogate the contents of the BTB. Because the Flush+Reload attack has a high signal-to-noise ratio, the combined technique is more precise than prior techniques for querying the branch predictor [6, 24, 25, 38, 45, 54],

To demonstrate the power of BunnyHop, we use it to reverse engineer the BTB of modern Intel Core processors. We reproduce results on the number of sets and tag folding from prior works[45, 75, 88]. We further show that the BTB consists of two substructures, one that stores only *short* branches, i.e. those where the target address is close to the source address, whereas the other stores all branches, resolving past discrepancy between [45] and [88] regarding BTB associativity. We identify the BTB replacement policy that the BTB is competitively shared between hyperthreads.

To demonstrate the implications on security, we design three attacks based on BunnyHop. The first two, BunnyHop-Reload and BunnyHop-Probe, are the BTB counterparts of the Flush+Reload [33, 84] and the Prime+Probe [49, 55] attacks respectively. Specifically, in the BunnyHop-Reload attack, the adversary first evicts a target prediction from the BTB and then interrogates the BTB to check whether the victim execution has reinstated the previously evicted prediction. In the BunnyHop-Probe attack, the adversary fills a BTB set with prediction and detects victim's activity by checking if the BTB set still contains the adversary's prediction. The third attack, BunnyHop-Evict, is a new attack that causes the victim to evict its own cache lines, allowing us to overcome previously proposed defenses against cache attacks.

We demonstrate the efficacy of BunnyHop-Reload, showing how to use it to break Kernel Address Space Layout Randomization (KASLR). The attack also demonstrates a novel use of the BTB aliasing to reduce the number of attack rounds required. We further demonstrate the use of BunnyHop-Reload to attack an implementation of an elliptic curve secp256k1 running inside an SGX enclave

We use BunnyHop-Evict to implement an Evict+Time attack against an AES implementation protected with both cache coloring [66, 86] and table preloading [55, 87]. Finally, We use BunnyHop-Probe to implement a cross-hyperthread attack on an implementation of RSA.

In summary, our paper makes the following contributions:

- We reverse engineer the instruction prefetcher, demonstrating that it follows branch prediction advice from the BTB (Section 3).
- Based on the operation of the instruction prefetcher, we develop a technique called BunnyHop, which encodes branch predictions as cache state, allowing us to use cache attack techniques for interrogating the BTB. We use BunnyHop to reveal the structure of BTB and resolve the discrepancy in past research regarding BTB associativity (Section 4).
- We present the BunnyHop-Reload attack, an instantiation of the Flush+Reload attack, targeting the BTB. We demonstrate the use of this new BunnyHop-Reload attack through stealing a secret information from an elliptic curve secp256k1 implementation running on SGX (Section 5).
- We further demonstrate another application of our BunnyHop-Reload through an attack on KASLR (Section 6).
- We present the BunnyHop-Evict attack, a combination of BunnyHop and Evict+Time attacks. BunnyHop-Evict is a confused deputy attack on the instruction prefetcher, which induces a victim to evict its own data from a cache. We demonstrate the use of BunnyHop-Evict through attacking a hardened implementation of AES 128 running in a kernel module (Section 7).
- We present the BunnyHop-Probe attack where we put together our BunnyHop attack and Prime+Probe attack. We show how to use our BunnyHop-Probe attack to tackle ASLR and recover the key from square-and-multiply-always RSA (Section 8).

The source code for BunnyHop is available at https://github.com/0xADE1A1DE/BunnyHop.

Responsible Disclosure. The results in this paper were reported to Intel. The company responded that the issue is covered by its software cryptography guidelines [22] and that no embargo period is required.

2 Background

Branch Prediction. The front end of modern CPUs is responsible for fetching, decoding, and feeding instructions to the execution engine. Execution paths that depend on branch instructions cannot be decided until the branch condition is resolved. Hence branch instructions can stall the pipeline. To keep the pipeline busy, the processor predicts the conditions and targets of the branch and speculatively executes the predicted path. When the condition or target is resolved, the processor verifies the prediction. In the case of a correct prediction, speculative execution bridges over the potential stall. If the prediction turns out to be wrong, instructions that were incorrectly executed are squashed and execution continues from the correct path.

The branch prediction unit (BPU) in the front end uses historical branch data to predict future branch outcomes. This historical data is recorded in several structures within the BPU. Particularly, in this work we are interested in the branch target buffer (BTB), which records the target address of branches.

Instruction Caching. To maintain a consistent stream of instructions, the front end must bridge the speed gap between the fast processor and the slower memory. One of the main techniques for bridging the gap is caching recently executed instructions in the level-1 instruction cache (L1-I). The L1-I interfaces with the rest of the memory subsystem, including the level-2 cache and the last-level cache (LLC).

Instruction Prefetch. While programs tend to exhibit significant locality, there are cases where instructions are not in the L1-I cache. To reduce the wait for such instructions, the instruction prefetcher brings memory locations predicted to contain future instructions into the L1-I cache. To predict future instructions, the instruction prefetcher relies on other

components such as the BTB [16, 21, 44, 70] or on execution history [27, 90].

Cache Structure. As caches play an important role in the microarchitecture, we now look at how they are implemented. Most modern caches are set associative. That is, the cache consists of multiple *sets*, which consist of multiple *ways*. Each element is mapped to a single cache set, but can be stored in any of the ways of the set. A *tag* identifies the element within a set. To check if an element is cached, the processor computes the set id, and searches for a tag match within the ways. If the tag matches, the processor can use the entry. Otherwise, the entry needs to be retrieved or recomputed.

Replacement Policy. Typically, when inserting an element to the cache, there is a need to replace another entry. The replacement policy of a cache determines which element is to be replaced. The least recently used (LRU) policy chooses the element in the set that was not used for the longest time. To reduce the resources required for implementing this replacement policy, many caches use a pseudo LRU (PLRU) policy that approximates LRU.

Cache Attacks. Because the state of caches depends on prior executions and, at the same time, affects the future execution speed, monitoring execution speed can reveal information on past execution. Various cache attacks [28] have been proposed in the past, targeting multiple caches, including data caches [49, 55, 84], instruction caches [7, 91], and branch prediction caches [6, 24, 59].

Prime+Probe. Prime+Probe [49, 55] is a cache attack that exploits contention on a cache set. In the attack, the adversary first fills all the ways in a cache set with their data. The adversary then waits for the victim to execute. Finally, the adversary measures the time to access the previously cached data. A short time indicates that the data is still cached, hence the victim did not access the data that maps to the same set. A long access time indicates that some of the previously cached data has been evicted, presumably due to contention with the victim.

Flush+Reload. In the Flush+Reload attack [33, 84], the attacker first evicts a victim entry from the cache. Later, the attacker attempts to access the victim entry. A fast access time indicates that the entry is cached, hence the victim has accessed it. Slow access indicates that the victim has not accessed the data.

Evict+Time. In the Evict+Time attack [55], the attacker first evicts an entry that the victim may use from the cache. The attacker then measures the time it takes the victim to execute an operation. A fast execution time indicates that the victim has not used the evicted entry.

Spectre Attacks. For decades, speculative execution was considered a harmless performance improvement. However, the discovery of Spectre [43] showed that it does have severe security implications. Specifically, Spectre observes that squashing mispredicted instructions does not revert the changes they made in the microarchitecture. Consequently, an attacker can force a branch misprediction to bypass software-based protection, access secret data, and transmit it through a microarchitectural channel, e.g. via the cache.

Spectre attacks differ in the type of prediction they abuse. Spectre-v1 exploits misprediction of conditional branches, i.e. whether the branch is taken or not. Spectre-v2 exploits indirect branches to speculatively execute arbitrary code within the address space of the victim.

Spectre-v2 Countermeasure. To protect against Spectre-v2, Intel proposed several countermeasures:

- Retpolines [30, 39] are a drop-in replacement for indirect branches that use a RET instruction instead of the indirect branch.
- Indirect Branch Prediction Barrier (IBPB) [10, 40] is a branch prediction barrier that prevents indirect branches that execute before the barrier from affecting branches that execute after the barrier.
- Single Thread Indirect Branch Predictors (STIBP) [10, 40] is a configuration option that prevents indirect branches on one hyperthread from causing speculative execution on the other hyperthread.
- Indirect Branch Restricted Speculation (IBRS) [10, 40] is a configuration option that prevent indirect branches executed at a low privilege (e.g. user mode) from affecting the prediction of branches at a higher privilege (e.g. kernel mode).

See Barberis et al. [11] for a more detailed discussion.

3 Reverse Engineering the Instruction Prefetcher

The instruction prefetcher forms part of the instruction fetch unit (IFU) of the processor [41]. To gain understanding on the operation of the prefetcher, we build on the observation that prefetching a memory location inserts it to the cache. We perform the analysis on multiple Intel processors. (See Table 1.) In the text we describe the results for a Core i7-10710U processor.

3.1 Prefetch Depth

The first question we answer is what the prefetch depth is, i.e. how many memory lines are prefetched ahead of the instruction pointer. For that, we use a function that consists of a single RET instruction, followed by unused memory. We flush memory lines following the RET instruction from the cache, call the function, and measure the time to reload the memory lines in the unused memory. To eliminate potential effects of the branch prediction unit on the instruction prefetcher [41], we invoke the Indirect Branch Predictor Barrier (IBPB) before calling the function.

We find that after calling the function, the 14 memory blocks following the RET instructions are typically in the

Model	Depth	Model	Depth
i7-2600S	1	i5-8265U	14
i5-3470	2	i7-9750H	14
i7-4770	7	i7-10710U	14
i5-5250U	7	i9-11900K	14
i7-6700	14	i9-12900KF (P)	26
		i9-12900KF (E)	8

Table 1: Instruction prefetcher depth in processor models.

cache, whereas blocks 15 and above are not in the cache. Hence, we conclude that the prefetcher depth is 14. We note that the depth varies between processor models. See Table 1 for complete details.

Because the flushed memory blocks are after a RET instruction, we know that their contents have not been architecturally executed. However, that does not guarantee that they have not been speculatively executed. An alternative explanation is that the instructions have been cached because the processor speculatively executed them. To rule out this explanation, we add a memory access after the RET instruction. We then perform a Flush+Reload attack on the memory location targeted by the memory access, finding that it is not cached when the function is invoked. Having confirmed that the code after the RET instruction is not executed, we conclude that the instruction must have been prefetched.

> ***Obs.* 1.** The instruction prefetcher prefetches multiple memory lines, with a model-dependent limit.

```
branch_train:                branch_probe:
  JMP T1                       RET
  NOP      (×1019)             NOP      (×1023)
T1:                          T2:
  RET
```

Listing 1: Testing Branch Instruction Prefetch.

3.2 Prefetching and Branches

Intel states that instruction fetching is helped by the BPU [41]. Indeed, we observe that without IBPB, repeated executions of our empty function do not show evidence of prefetching.

To analyze the effects of branches on prefetching, we rely on branch shadowing [24, 45, 64], a technique that exploits aliasing between branches at addresses that share the least significant 30 bits.

We use AssemblyLine [1] to create two functions `branch_train` and `branch_probe`, shown in Listing 1. `branch_train` consists of a JMP instruction that jumps 1019 bytes forward before returning. We note that the length of the JMP instruction is five bytes, hence `T1` is at an offset of

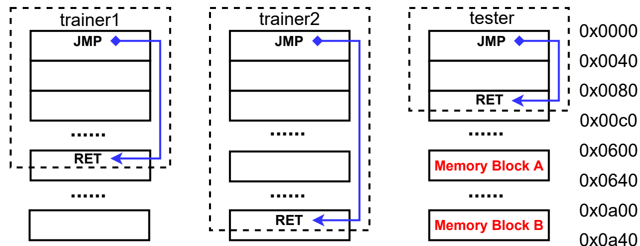

Figure 1: Memory layouts of the function `trainer1`, the function `trainer2` and the function `tester`.

1024 bytes from the start of `branch_train`. The function `branch_test` is an empty function with a single RET, similar to the one used in Section 3.1, labeled `T2` at an offset of 1024 bytes from the start.

We instantiate the functions at aliased addresses, i.e. addresses that have the same 32 least significant bits. Note that because the functions are at aliased addresses, `T1` and `T2` are also aliased. We invoke `branch_train` and then `branch_probe` before testing for prefetched memory lines. We find that although `branch_probe` does not perform any branch, the memory line at `T2` is prefetched, as well as the 13 subsequent memory lines.

We then change `branch_train` to execute a chain of branches. We find that the prefetcher follows these branches, even if the code of `branch_probe` does not include them, without affecting the prefetch depth.

> ***Obs.* 2.** The instruction prefetcher follows trained branches, irrespective of the code in the prefetched memory.

3.3 Prefetching and BPU collisions

We now turn our attention to understanding how aliased branches affect each other. For this, we add a second trainer and check the interaction between the two trainers. An example of such a setup appears in Figure 1, where the first trainer, `trainer1` jumps to offset 1536 (0x600), the second, `trainer2`, jumps to offset 2560 (0xa00), and the tester branches to offset 128 (0x80). We instantiate all three functions at aliased addresses, execute them in order (i.e. first `trainer1`, then `trainer2`, and finally `tester`) and check which memory block is prefetched. We find that invariably the instruction prefetcher follows the latest aliased branch. That is, in the example in Figure 1, memory block B is prefetched, whereas memory block A is not.

We then test `trainer1` and `trainer2` with different branching instruction types, including conditional branches, indirect branches, calls and return instructions. When changing the code, we make sure that we maintain the branch source address (i.e. the address of the instruction following

the branch) and the branch destination address. We further ensure that conditional branches are taken. We find that for direct branches memory block B is always prefetched. However, for indirect branches in some cases memory block A is prefetched and in other memory block B. This agrees with the claim that the branch prediction unit can predict multiple destinations for indirect branches [88]. We leave the task of determining how the instruction prefetcher chooses between the potential destinations to future work.

When `trainer2` uses a conditional branch that is not taken, the results become more complex. If the offsets of `trainer1` and `trainer2` are the same, the instruction prefetcher follows the training from `trainer1`. However, if the offsets are different, the instruction prefetcher *sometimes* follows the training from `trainer1`, i.e. prefetches memory block A. In other cases it forgets the training and does not prefetch either memory block A or B. Moreover, if we replace the JMP instruction in `trainer2` with non-branch instructions, such as a sequence of NOP instructions, it forgets the training and does not prefetch either memory block A or B.

> ***Obs.* 3.** Direct branches replace the prefetcher prediction, indirect branches may replace the prediction, and non-branch instructions may delete existing predictions.

3.4 Prefetching and Hyperthreading

Previous works show that the branch predictor is shared between hyperthreads [24, 72]. To test whether the prefetcher is also shared, we execute `trainer1` from Figure 1 on one hyperthread, and then `tester` on the second hyperthread of the same core. We find that memory block A is never prefetched and conclude that there is no training across hyperthreads.

> ***Obs.* 4.** There is no cross-hyperthread training of the instruction prefetcher.

3.5 Prefetcher Operation

```
op_train:                 op_probe:
  RET                       NOP    ×n
                            RET
```

Listing 2: Testing Instruction Prefetch on hyperthreads.

To better understand how the instruction prefetcher progresses over time, we test how prefetching depth is affected by temporal restrictions. For that, we use the code in Listing 2. We first invoke the function `op_train`, which trains the prefetcher not to prefetch subsequent memory blocks. We then run `op_probe`, which consists of a sequence of NOP instructions followed by a RET.

The reasoning behind this arrangement is that when executing `op_probe`, the prefetcher initially follows the training. However, at some stage the first NOP instruction is decoded. This instruction does not match the prediction, resulting in a prefetcher resteer to prefetch subsequent memory lines. This prefetch direction continues until the RET is decoded, resteering the prefetcher again. By changing the number of NOP instructions in `op_probe` we control the time between the first resteer and the decoding of the RET.

We vary the number of NOP instructions between 0 and 50 and test under two conditions. First we test on one hyperthread when the other hyperthread is idle, then we test again, but with the other hyperthread active. Figure 2 shows the results.

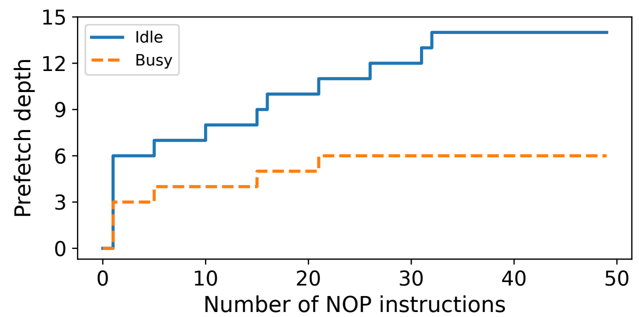

Figure 2: Prefetch depth as a function of the number of NOP instructions in `op_probe` on one hyperthread, when the other hyperthread is idle or busy.

As expected, when the number of NOPs is 0, there is no prefetch. However, when adding a single NOP, the prefetch depth jumps to six for the idle case and three for the busy. The prefetch depth then increases with the number of NOPs.

We know that the processor fetches code in blocks of 16 bytes and can decode up to five instructions per cycle [41]. Observing the way the prefetch depth changes with the number of NOPs for the idle case, we see a depth increase every five NOPs, and a further increase whenever crossing a 16-byte boundary. Thus, we conclude that the prefetcher prefetches one memory block per cycle. We believe that the jump to depth six with one NOP instruction is due to the L1 cache latency (four cycles) and the time to decode and resteer the prefetcher.

Observing the busy line, we see that the prefetch depth is roughly half that of the idle case. This agrees with having a single prefetcher that alternates between the hyperthreads.

> ***Obs.* 5.** The prefetcher prefetches one cache line per cycle, alternating between the two hyperthreads if both are active.

Model	i7-4770	i7-6700 i5-8260U i7-9750H i7-10710U	i9-11900K
Capacity	4,096	4,096	5,120
Set index	[12:4]	[13:5]	[13:5]
Ways (S+L)	4+4	4+4	6+4
Tag	[30:22]⊕[21:13]	[29:22]⊕[21:14]	[33:24]⊕[23:14]
Short bits	[9:0]	[9:0]	[11:0]
Replacement	PLRU	Alt.+LRU	UPLRU

Table 2: BTB information for processor models.

4 Reverse Engineering the BTB

So far, we have studied the behavior of the instruction prefetcher. In this section we use the instruction prefetcher to study the structure of the branch target buffer (BTB). For the investigation we use our proposed BunnyHop technique, which is based on the observation that we can use the instruction prefetcher to transfer state from the BPU to the cache. Specifically, when executing a NOP instruction at an address that has a prediction for a branch, the instruction prefetcher will prefetch the target of the predicted branch, inserting it into the cache.

Similar to past works [45, 75, 88], to reverse engineer the BTB, we first invoke the function `btb_train` from Listing 3 (left) to create an entry in the BTB. The function executes a JMP instruction with a predefined offset, thus creating a BTB entry indicating that the address of the JMP instruction contains the branch. We then execute a sequence of test branches that we wish to test whether it removes the entry that `btb_train` created.

To check if the test branches evicted the entry, we invoke the function `btb_probe` from Listing 3 (right). The functions `btb_train` and `btb_probe` are aliased hence if the entry is still in the BTB when `btb_probe` is invoked, the instruction prefetcher will follow the entry and prefetch the predicted branch target. Finally, we use the Flush+Reload technique to test if the predicted target address is cached. If it is, we determine that the test branches did not evict the entry that `btb_train` created.

```
btb_train:                    btb_probe:
  JMP T1                        NOP   ×32
  NOP ×skip                     RET
T1:
  RET
```

Listing 3: Code for Reverse Engineering the BTB.

We follow the approach of past works [45, 75, 88], replacing their detection method with our technique. Where these works agree, we confirm their findings, including that the BTB is set-associative, what bits are used to select the set, and the tag function. (See Table 2 for a summary.) Moreover, we identify an additional structure inside the BTB, reverse engineer its replacement policy, and clarify the interaction between the BTB and the branch history buffer (BHB). We now describe the experiments that uncover these results. As earlier, results in the text refer to Core i7-10710U. See Table 2 for other processor models.

4.1 Long and Short Branches

One issue where the works of Lee et al. [45] and Zhang et al. [88] disagree is the associativity of the BTB. The former specifies associativity of four, whereas the latter specifies eight. To test the associativity, we use a branch offset of 2048 for our `btb_train`. We further make sure that the addresses of `btb_train` and `btb_probe` agree on bits [31:0]. For test branches we use JMP instructions at addresses that share the 25 least significant bits of `btb_train`, but have a different value at bits [29:26]. Observe (Table 2) that bits [13:5] determine the BTB set, hence all of the test branches fall in the same BTB set as `btb_train`. Moreover, the tag is determined by folding bits [29:14], i.e. the bitwise XOR of bits [29:22] and bits [21:14]. Hence, the test branches have different tags, which are also different from the tag of `btb_train`, so they are not aliased.

We first use test branches with an offset of 2048 bytes. When using less than four branches, the entry created by `btb_train` is never evicted. However, with four or more test branches invoking `btb_probe` no longer prefetches the predicted target, indicating eviction of the entry. This result agrees with the claim of Lee et al. [45] that the associativity of the BTB is four. We then use test branches with an offset of 512 bytes and find that four branches are not enough and we need eight test branches to evict the entry created by `btb_train`, as specified in Zhang et al. [88].

Having determined that there are two types of branch offsets, we experiment with combinations of branch addresses and offsets to find out how the processor decides the type of the branch. We find that if the address of the last byte of the JMP instruction and the target address of the branch share all but bits [9:0], the branch is considered "short", and the associativity of the BTB is eight. Otherwise, the branch is "long" and the associativity is four.

***Obs.* 6.** The BTB supports two branch types, short and long.

When we use a short branch for `btb_train`, we find that eight short test branches always evict the entry created by `btb_train`. However, when using long test branches with a short training branch, in some cases as few as four long branches evict the short training branch, whereas in others 20 long branches fail to evict the training branch.

This behavior of short and long branches is consistent with the variable-size BTB (VS-BTB) of Hoogerbrugge [37]. In

a VS-BTB, a BTB set contains two types of entries. Long entries can store all branches, whereas short entries can only store short branches.

> ***Obs.* 7.** The processor appears to use a variable-size BTB, with four long ways and four short ways.

4.2 Predicted Branch Address

One of the main aims of the VS-BTB design is to reduce the number of bits stored in the BTB by using fewer bits to store short branches. Specifically, for short branches, the VS-BTB only stores some of the least significant bits of the target address. With this structure, replacing the least significant bits of the source address with the bits stored in the BTB produces the target address. In this section we aim to find how many target address bits are stored in the BTB.

We create random pairs of `btb_train` and `btb_probe` at aliased addresses, i.e. where the addresses match on bits [13:0] and on the (folded) tag in bits [29:14]. Given a guess that the BTB stores k bits of the target address, we compute the predicted branch address for each pair assuming the guess is correct. That is, if `btb_train` jumps to target address t and `btb_probe` is at address p, we generate the predicted address r such that $r[46{:}k]= p[46{:}k]$ and $r[k-1{:}0]= t[k-1{:}0]$. We then invoke `btb_train`, flush the guessed predicted target address from the cache, invoke `btb_probe`, and test whether the predicted target address is prefetched.

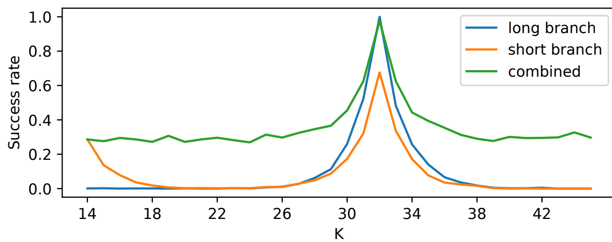

Figure 3: Prefetch probability for target bit guesses.

We perform the experiment once when `btb_train` uses a long branch and then when it uses a short branch. For each guess of the number of target address bits we calculate the probability that the predicted target address is prefetched. As Figure 3 shows for long branches we achieve an almost perfect prediction for a guess of 32. Hence we conclude that long ways store 32 bits of the target address.

When using a short branch in `btb_train`, we see that the probability of detecting a prefetch also peaks at a guess of 32 bits, but the probability is significantly lower than that of long branches. We hypothesize that the reason is that our predicted target branch only matches if the short branch is stored in a long way of the BTB. To test the hypothesis we repeat the experiment, but this time we combine the success rate for a guess of k bits with the success rate for a guess of 10 bits. That is, we mark a test as successful if we find evidence of prefetch for either guesses. As we can see in the figure, the combined guess has a baseline success rate of about 50%, achieving almost perfect success at $k = 32$. Thus, we conclude that in our experiment, about 50% of the short branches fall in short ways, which store 10 target address bits.

> ***Obs.* 8.** Long BTB ways store 32 bits of the target address. Short ways store 10.

4.3 BTB Replacement Policy

We now turn our attention to the replacement policy used in the BTB. For that, we use the approach of Abel and Reineke [2]. That is, we perform several `btb_train` functions that all fall within the same BTB set. We then use the corresponding `btb_probe` functions to determine which training remains in the BTB. We now describe the replacement policies we have identified.

Core i7-6700. Recall that the processor uses a variable-size BTB with four short ways and four long ways. To recover the replacement policy, we first focus on long branches that can only be stored in long ways. We then look at short branches that can be stored in both long and short ways. Finally, we identify the interaction between the two branch types.

Following the procedure of Abel and Reineke [2] with long branches, we find that long ways have an LRU replacement policy. Repeating with short branches, we find that the eight ways are divided into two banks of four ways each. Within each bank, the processor uses an LRU replacement policy. To decide which bank is used for replacement, the processor tracks the least recently used bank, and in the case of a replacement will use LRU within that bank.

To determine whether the banks correspond to long and short ways, we create eight pairs of short `btb_train` and `btb_probe` functions, such that all functions are in the same BTB sets. The functions in the pair also have the same tag, but their addresses differ on some bits in [31:14]. Recall from Section 4.2 that in such a case, the prefetched address will depend on whether the branch is stored in a short way or in a long way. We then invoke the trainer functions one by one, and use the probes to test whether the trainers are stored in long or short ways. We find that the branches alternate between long and short ways. We conclude that the banks we identified correspond to short and long ways.

In summary, the BTB consists of two banks, one for short and the other for long ways. Each bank has an LRU replacement policy. Replacement for short branches chooses the least recently used bank, following LRU within the bank. We call this policy “Alternating LRU”, denoted as Alt.+LRU in Table 2.

Core i9-11900K. We first identify the BTB of the Core

i9-11900K processor has four long ways. Following the procedure of Abel and Reineke [2] we find that the replacement policy for these is tree-PLRU.

Experimenting with combinations of short and long branches, we find that the BTB supports six short ways. The replacement policy used for short branches is tree-PLRU on a tree that includes both the short and the long ways. As the tree, which is depicted in Figure 4, is not balanced, we call this policy “unbalanced PLRU”, noted as “UPLRU” in Table 2.

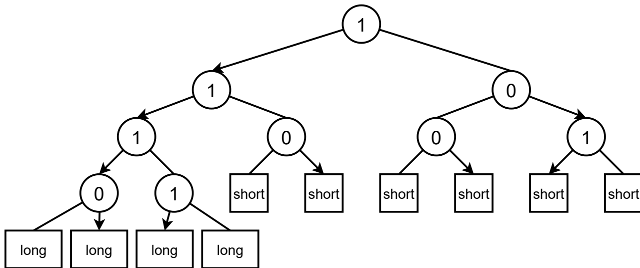

Figure 4: Tree of unbalanced PLRU (Core i9-11900K)

5 The BunnyHop-Reload Attack

So far we have demonstrated how the instruction prefetcher can be used to reverse engineer the BTB. In this section we start looking at the security implications of the instruction prefetcher. We demonstrate BunnyHop-Reload, the branch-prediction equivalent of the Flush+Reload attack [84].

Recall that in a Flush+Reload attack, the adversary first evicts a memory location from the cache. The adversary then waits before testing whether the memory location is cached, indicating that the victim has accessed it. BunnyHop-Reload follows a similar sequence, aiming to detect whether the victim has taken a specific branch. For that, the adversary first evicts the BTB entry of the victim branch by executing a NOP instruction at an aliased address. The adversary then waits for the victim to execute. Finally, it exploits the BunnyHop effect to test whether the victim has taken the branch. That is, the adversary executes a NOP instruction at an aliased address then checks whether the instruction prefetcher prefetched the target of the branch.

To demonstrate the effectiveness of BunnyHop-Reload, we use it to recover secret information from a vulnerable implementation of the elliptic curve secp256k1, running inside an SGX enclave. We first provide background on SGX and describe the victim and the attack setup. We then describe the attack and the experimental results.

SGX. Intel Software Guard Extensions (SGX) is an instruction set extension that provides a secure execution environment, called an *enclave*, which is protected against strong adversaries, including those that control the operating system. While enclaves are isolated from other code which executes

Algorithm 1: wNAF Algorithm:

Input : scalar d in wNAF d_0, ..., $d_{\ell-1}$ and precomputed points $\{G,\pm[3]G,\pm[5]G,...,\pm[2^w-1]G\}$
Output : $[d]$G

```
1  Q ← ?
2  for j from ℓ−1 downto 0 do
3      if j ≠ ℓ−1 then
4          Q ← point_double(Q)
5      end
6      ...
7      if d_j ≠ 0 then
8          Q ← point_add(Q, [d_j]G)
9      end
10     ...
11 end
```

on the processor, they do share the use of microarchitectural components, allowing adversaries to mount side-channel attacks [18, 38, 45, 47, 51, 52, 59, 59, 69, 76]. Moreover, past research has demonstrated that including the operating system in the threat model allows for very strong adversaries [76, 77].

Victim. Our victim code is the implementation of the secp256k1 elliptic curve found in OpenSSL version 1.1.0h. The implementation uses the wNAF algorithm (Algorithm 1) for performing scalar multiplication over an elliptic curve. In a nutshell, wNAF represents a scalar as a sequence of digits that can be either 0 or odd values between -2^w+1 and 2^w-1 for an implementation-dependent window size w. The algorithm scans the scalar performing an elliptic curve point double operation for every digit, and an elliptic curve point addition for each non-zero digit. Like prior attacks on the secp256k1 curve [12, 26, 58], our attack aims to find the positions of the non-zero digits.

Experiment Setup. We conduct the experiment on an Intel Core i7-10710U, with 6 cores, running Ubuntu 20.04. As in other SGX attacks [20, 47, 69, 76, 77, 78], we disable the Intel SpeedStep and Turbo Boost technology. Furthermore, we disable address space layout randomization within the enclave. We use SGX-step [76] to single-step the enclave.

Attack. We use AssemblyLine [1] to construct a spy function that consists of six NOPs followed by a RET. We instantiate the function at an address the shares the 32 least significant bits with the branch at Line 7 of Algorithm 1. Executing the spy function achieves two aims. First, as described in Section 3.2, in case there is a branch training for the victim branch, executing the spy will prefetch the target address of the branch prediction. Second, as seen in Section 3.3 because the NOP instructions are not branching, executing them deletes the training of aliased branches from the BTB.

For the attack, we use SGX-step to single-step the enclave. After executing an enclave instruction, we test for two

events. The first event is the execution of code in the function `point_double()`, invoked at Line 4 of Algorithm 1. The second event is taking the monitored branch at Line 7.

To test the `point_double()` function, we mark the page where it resides as not accessed before each victim function step. Executing the code in the function would mark the page state as accessed, which we can detect after executing a single victim instruction.

To monitor the branch, we use BunnyHop-Reload. That is, we calculate the predicted branch target address based on the known branch offset of the branch at Line 7 and the known address of our spy function. That is after SGX-step interrupts the enclave, we flush the predicted target from the cache, and invoke the spy. Due to the BunnyHop effect, if the victim has taken the victim branch, the instruction prefetcher will prefetch the predicted target, allowing us to test whether it is cached. Executing the spy also removes any training, setting up the state for executing the next instruction.

Experiment Result. Monitoring the execution of `point_double()` allows us to track the digit index. We combine this with the results of the BunnyHop-Reload attack to detect whether the digit is zero or not.

We randomly generated 25 pairs of public and private keys, for each pair of keys, we run the attack once achieving a success rate of above 98%.

6 Exploiting Target Leak

One of the main differences between the Flush+Reload attack and BunnyHop-Reload is that the former only reveals whether instructions have been executed (as executed instructions will have been cached) while the latter additionally reveals part of the target address for branch instructions. In this section we show how we can exploit this information to build a novel and efficient attack on kernel address space layout randomization (KASLR).

Cache attacks are divided into contention-based (e.g. Prime+Probe) and reuse-based (e.g. Flush+Reload). Aliasing in the BTB blurs the difference between those. Performing an aliased jump both replaces the existing entry, similar to Prime+Probe, and creates a new entry that the attacker can use, similar to Flush+Reload. Jump over ASLR [24] exploits the contention aspect. Our attack exploits the new entry created by the victim. Compared with Jump over ASLR, our method only needs to invoke the system call once. Like our attack, the RBTBP attack [54] exploits the new entry created by the victim. However, as Oliviera and Dutra [54] state, RBTBP cannot bypass Spectre-v2 mitigations.

Attack Overview. The aim of KASLR is to hide kernel addresses from adversaries to protect against code injection attacks. The Linux implementation of KASLR randomizes bits [29:21] of the base address of the kernel. Our aim in this attack is to recover the values of these bits. We note that recovering any known address in the kernel is sufficient for breaking KASLR because the offset from the base is fixed.

Recall that in Section 4.2 we show that the BTB contains both short and long ways. Long ways are capable of storing long branches and store bits [31:0] of the target branch address. Thus, if we can recover the target of a known branch in the kernel, we completely break KASLR.

Thus, for the attack we choose a long branch in the kernel, and invoke a system call that takes the branch. We then create an aliased function in user space, which when executed will prefetch the predicted branch target address. An aliased function contains multiple NOPs and a RET. The first NOP in the function is aliased with the kernel branch. We then test all possible target addresses to see which has been prefetched, identifying bits [31:0] of the kernel branch target.

Creating an Aliased Function. To create an aliased function, we need to know the tag of the targeted kernel branch. Recall that the tag is created from folding bits [29:14] of the branch address. Because KASLR randomizes bits [29:21], we do not know what the tag is. However, as there are only 256 different tags, we can brute force the tag by generating 256 functions, each matching one branch address.

We do need to take care that invoking a large number of function does not evict the target branch from the BTB. We only need a single CALL instruction for invoking all of the functions, and can choose its address so it does not use the same BTB set as the victim branch. Moreover, each of our aliased function contains 32 NOP instructions, forcing the subsequent RET into the next BTB set.

Experiment Setup. Our attack targets the `kill` system call. The `kill` system call takes an integer parameter `pid`, indicating the ID of the process that receives the signal, and invokes the function `kill_something_info`. In the case that `pid` is less than 1 and is not -1 or `INT_MIN`, the function `kill_something_info` uses a CALL instruction to call `__kill_pgrp_info`, which we target.

We run the experiment on Ubuntu 20.04 LTS with the kernel version 5.13.0-52-generic. We set the kernel command line to enable all of the Spectre-v2 defenses by adding the parameter `spectre_v2=on`.

Evaluation. We test possible loaded addresses after each execution of an aliased function. We test our attack on several processors with 10 random KASLR offsets (10 reboots) and each offset 10000 times. In about 5.78% of the attempts fail to obtain results and have to repeat the attack. After repeating (if necessary), the attack recovers the correct offset in almost all cases. Detailed accuracy rates are summarized in Table 3.

Comparison with Jump over ASLR. To compare against prior attacks, we implement the attack from Jump over ASLR [24] on multiple processors. On Skylake and newer processors, the attack fails because the timing difference is not significant enough to filter out the branch collision. On the Haswell machine, we can reproduce the Jump over ASLR

Model	Accuracy
i7-6700	100.00%
i5-8265U	99.92%
i7-9750	99.98%
i7-10710U	99.94%

Table 3: Accuracy of breaking KASLR with BunnyHop-Reload.

attack, where it takes around 60 milliseconds and has close to 100% accuracy. Note that our attack has similar accuracy but is significantly faster than Jump over ASLR.

7 The BunnyHop-Evict Attack

Cache-based timing attacks can be broadly classified by the type of information measured by the attack. Time-driven attacks measure victim process execution time to infer the presence or absence of cache collisions, and in turn infer the memory access patterns of the victim process [4, 28, 53]. Time-driven attacks can be mitigated through preloading [15, 55, 87], a mitigation technique that masks delays in program execution time that are caused by memory access patterns that miss the cache. Preloading loads program memory into the cache before it is used to specifically avoid memory access patterns that miss the cache.

In contrast, access-driven attacks measure and often manipulate the state of the cache (the presence or absence of specific cache lines in the cache) to infer memory access patterns of the victim process. Access-driven attacks can be mitigated through page coloring, a technique which separates security domains within the cache such that an attacker and their victim cannot influence the cache state of the other [66, 86, 89].

In this section we introduce BunnyHop-Evict, a confused-deputy attack [36] in which the instruction prefetcher is induced into evicting the victim's data from the cache instead of just prefetching the victim code. Specifically, to cause eviction, we train the branch predictor so that when the program executes a target instruction, the prefetcher prefetches a sequence of memory locations that form an eviction set. As we have seen in Section 3.1, prefetching brings the prefetched location into the cache. This causes contention on the cache set, resulting in an eviction of a target memory line. We harden the table-based implementation of AES in OpenSSL to support both preloading and page coloring. We then use BunnyHop-Evict to mount an attack on the hardened implementation and recover the last-round encryption key, completely bypassing both mitigations.

7.1 Overview

Listing 4 shows a proof-of-concept target, an AES 128 kernel module that presents an API for user-space processes to encrypt messages with a secret key. The implementation is based on OpenSSL 0.9.8b which uses a T-table lookup during the last-round of encryption. To mitigate time-driven cache attacks, we apply preloading on Lines 6 to 10. To mitigate access-driven cache attacks, we implement page coloring and ensure that the user-space processes and our kernel module use separate colors.

```
1  ssize_t device_read(char * buffer) {
2  copy_buffer_to_aes_input();
3
4  // Pre-load T-tables
5  for (i = 0; i < 16; i++) {
6    *(volatile u32*)&Te0[i * 16];
7    *(volatile u32*)&Te1[i * 16];
8    *(volatile u32*)&Te2[i * 16];
9    *(volatile u32*)&Te3[i * 16];
10   *(volatile u32*)&Te4[i * 16];
11 }
12
13 // Stall the execution until
14 // the finish of pre-loading
15 memory_barrier
16
17 AES_encrypt(input, output, &aeskey);
18
19 copy_aes_output_to_buffer();
20 return 0;
21 }
```

Listing 4: AES Kernel Module

Attacking the Victim. The key operating principle behind BunnyHop-Evict is that an attacker can induce the victim into prefetching an address of the attacker's choosing at a specific point in the program execution. We poison the victim so that when Line 17 is executed the prefetcher will prefetch memory to the cache, partially evicting the T-table and exposing the victim to a typical time-based cache attack that can recover the last-round key [53]. We choose Line 17 because this is after the victim has loaded prefetched the entire T-table into the cache but before they have started executing the encryption. We accomplish this task by extending the techniques from Section 3, where we show how a process can control the BTB to control which addresses are prefetched by the prefetcher.

Experiment Setup. We perform these experiments on an Intel i7-6700 and an Intel i5-8265U, running Ubuntu 20.04 with all Spectre countermeasures enabled.[1] The kernel is patched to use page coloring to isolate kernel processes from user processes in the cache. Moreover, to prevent concurrent attacks on core-local caches, we disable hyperthreading.

[1]Retpoline, IBPB, IBRS_FW, STIBP, and IBRS enabled via the `spectre-v2=on` kernel parameter.

7.2 Attack Description

The attack takes place over three distinct steps: *BTB Poisoning*, *Victim Execution*, and *Key Recovery*.

BTB Poisoning. First, the attacker aims to poison the BTB so that the prefetcher will prefetch enough memory to evict part of the preloaded T-table from the cache. The attacker executes a function consisting of a chain of fourteen direct jumps such that the virtual address of the first direct jump shares bits [30:0] with the instruction on Line 17 and the virtual addresses of the remaining branches share bits [11:0] with the virtual addresses of `Te4[0]`. Since the L1, and L2 caches are virtually indexed, addresses that share bits [11:6] are mapped to the same L1 and L2 cache set. Although the last-level cache is physically indexed, on a page colored system, virtual addresses that share bits [11:6] are also mapped to the same last-level cache set.

Victim Execution. The attacker then executes `device_read` through the */dev* file system API. On Line 2 the victim copies the plaintext provided by the attacker into a buffer managed by the AES algorithm. Lines 5–10 preload the entire T-table into the cache. Then, in Line 15 the victim waits until preloading completes. Finally, in Line 17 the victim begins executing the AES encryption algorithm. At this point, the prefetcher queries the BTB and finds the poisoned entry from earlier. The prefetcher prefetches memory addresses selected by the attacker, which map to the same cache set as `Te4[0]` thereby evicting it from the cache. Note that the prefetched memory blocks are cached in all of the cache levels in the hierarchy, including the L1-I instruction cache, and the unified L2 and LLC caches. At some point, the processor decodes the instructions on Line 17 and executes the rest of the AES encryption algorithm, where, during the last round, accesses to the `Te4[0]` can influence the execution time of the victim. Finally, on Line 19 the victim copies the now encrypted ciphertext back to the attacker process.

Key Recovery. During this whole process, the attacker measures the total execution time of `device_read`. Since the attacker induces the victim to evict `Te4[0]` from the cache, any access to entries that share the same cache line as `Te4[0]` must be served by system memory thereby increasing the total execution time of the encryption.

7.3 Experimental Results

We repeat the attack 35,000 times with randomly generated plaintexts and collect both the ciphertext and execution time for each. We then compute the Pearson correlation between the collected timing and targeted key byte guesses. That is, we first determine whether the evicted table entry is accessed for each combination of a key byte guess and ciphertext then calculate the correlation between the predicted execution time and the real execution time of the victim. Figure 5 shows how the correlation changes with the number of ciphertexts.

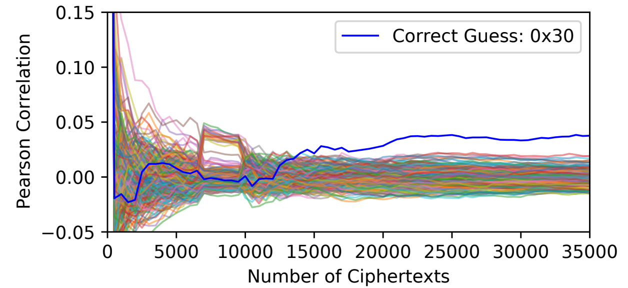

Figure 5: Pearson correlation for guesses for the first key byte, showing positive correlation for the correct guess 0x30.

The X axis shows the number of ciphertexts, whereas the Y axis shows the Pearson correlation. Each line corresponds to one key guess. As we can see, the lines for most key guesses converge towards zero, indicating no correlation. However, when the number of ciphertexts grows above 15,000, one line corresponds to key byte guess 0x30, showing an evidence of a positive correlation. Comparing to the ground truth, we find that value of the corresponding key byte is indeed 0x30.

8 The BunnyHop-Probe Attack

Cache attacks can be classified into contention-based and reuse-based attacks. So far, we have demonstrated BunnyHop-Reload, a reuse-based BunnyHop attack, to infer the contents of the BTB on the same thread. In this section, we demonstrate BunnyHop-Probe, a contention-based BunnyHop attack to infer the branch status of a sibling thread. BunnyHop-Probe primes a BTB cache set, waits for the victim process and then probes the BTB cache set. In the prime phase, the attacker primes an entire BTB cache set with direct branches. The attacker then exploits the known replacement policy (Section 4.3) to identify the eviction candidate, i.e. the element that will be predicted on the next miss in the target cache set. In the probe phase the attacker uses BunnyHop to test if the eviction candidate is still in the BTB. We first evaluate the feasibility of BunnyHop-Probe with a toy example, then we mount an attack on square-and-multiply-always implementation of RSA from GnuPG 1.4.14, which was proposed ads a countermeasure for the Flush+Reload attack [84].

8.1 Toy Example

We evaluate the feasibility of BunnyHop-Probe on several platforms with an artificial example. The code is listed in Listing 5. The victim processes a secret byte bit-by-bit and a branch is taken if the bit is one (line 8). The spy process creates shared memory of victim's binary and uses a Flush+Reload technique to detect when the `mem` is accessed by the

Model	Accuracy
i7-6700	99.13%
i5-8265U	93.25%
i7-9750	87.13%
i7-10710U	91.88%

Table 4: BunnyHop-Probe accuracy.

victim (line 4). The spy process starts BunnyHop-Probe after the mem is accessed by the victim. To evaluate the accuracy, we collect 25 samples that observe all eight accesses to the mem for each byte and we repeat the procedure for 100 randomly picked bytes. The result is listed in Table 4. The BunnyHop-Probe accuracy is close to 100% on the skylake machine while on newer machines the accuracy is relatively lower.

```
victim:
repeat 8 times:
  wait();
  memory_barrier;

  access(mem);
  memory_barrier;
  wait();
  secret(s);
  memory_barrier;
```

```
spy:
  for (;;)
  {
    flag = accessed(mem);
    if (flag)
    {
      prime_BTB_cache();
      wait();
      probe_BTB_cache();
      memory_barrier;
      check_result();
    }
  }
```

Listing 5: Two functions are executed on two hyperthreads. The spy process contains an infinite loop to BunnyHop-Probe the victim branch.

8.2 Attack Overview

The attack model assumes that a spy process is running on a thread while the victim is decrypting messages with RSA 4096 bits on the sibling thread. They spy process is running synchronously with the victim process.

Square-and-multiply-always RSA. To mitigate the Flush+Reload attack of Yarom and Falkner [84], GnuPG 1.4.14 implements a square-and-multiply-always exponentiation algorithm. Different from the square-and-multiply algorithm, which executes the multiplication only if the key bit is one, the square-and-multiple-always algorithm constantly executes a multiplication but it discards the result if the bit is zero. The GnuPG implementation uses a conditional branch to test whether the bit is one or not. It is this key-dependent branch that we target. The branch is recorded in the BTB if the bit is one. By observing the existence of the branch in the BTB, the attacker can infer the key bit.

Tackle ASLR. Address Space Layout Randomization randomizes all virtual address bits except the page offset bits, normally bits [11:0]. The BTB is indexed with bits [12:4] or [13:5] according to Table 2. With the deployment of ASLR, an attacker needs to guess one or two bits of the BTB set bits which leaves two or four possible BTB sets respectively. An attacker only needs to always prime the same BTB set and repeat the attack until clear signals are obtained. Statistically, the probability of priming the correct BTB set would be 25% or 50%, depending on the BTB index bits.

Experiment Setup. We perform the experiment on Intel i7-6700, running Ubuntu 20.04 with Spectre-v2 mitigations. In the experiment we assume the attacker always primes the BTB cache set that holds the key-dependent branch.

8.3 Attack Description

The attack consists of three steps: *BTB Priming*, *BTB Probing*, and *Key Recovery*.

BTB Priming. The attacker primes the BTB cache set with eight short branches. Depending on the page offset of the key-dependent branch, the attacker needs to pick the branch offset carefully such that the branch address and branch target address share all the bits except bits [9:0] according to Table 2. To prime a BTB set and set the LRU state of the set, the attacker executes all eight short branches and then selects a branch to be least recently used.

BTB Probing. After priming the BTB cache set, the attacker waits for a short period and then probes the BTB cache set. Relying on the disclose of the BTB replacement policy, the attacker probes one BTB slot the least recently used with a NOP instruction. The attacker infers the existence of the least recently used branch in the BTB with BunnyHop-Reload technique. If the key-dependent branch is executed by the victim, the least recently used branch will be evicted and the instruction prefetcher will not fetch the target memory block.

Key Recovery. The attacker repeats the attack until enough valid results that attacker primes the correct BTB cache set are collected. Due to the adjustment of CPU frequency, the probe results are not necessarily fall in the same Prime+Probe slot among all repeated attacks. By analyzing multiple results, the attacker considers the key-dependent branch is taken in one Prime+Probe slot if the branch is taken in this slot with over 40% possibilities among all results.

8.4 Experimental Results

In the experiment, the key-dependent branch is located at 0x56e, and thus we construct short branches with JMP 640. Between the prime and probe procedure, we wait for 50,000 cycles. Between two Prime+Probe slots that observe the key-dependent branch, eight Prime+Probe slots are not detecting any key-dependent branch executions. We collect 100 samples

and consider the branch is taken in a Prime+Probe slot if the branch is taken in this slot over 40% of the samples and we are able to identify the first half key bits with 100% success rate. Figure 6 shows partial results, a taken branch suggests that the key bit is 1; otherwise it is 0 and thus the recovered bits are 0100111011101011010111101.

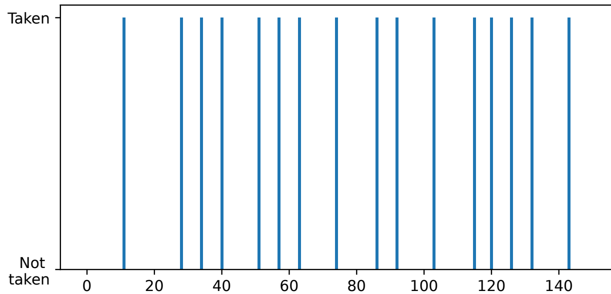

Figure 6: Prime+Probe result.

9 Countermeasures

In this section we explore possible countermeasures for BunnyHop attacks.

BunnyHop-Reload and BunnyHop-Evict. The main cause of both BunnyHop-Reload and BunnyHop-Evict is that the processor allows cross-domain branch training. One approach to prevent such cross-domain training is to associate the ID of the security domain with the BTB entry, and allow training only if IDs match. X86 processors already support address space IDs (ASIDs) for the translation lookaside buffer. Extending this to the BTB is, therefore, a possible solution.

Alternatively, wiping the BTB state during a context switch can also protect against the attack. For example, the operating system can use IBPB during context switches or when switching between user and kernel space. However, this approach does not protect against the SGX attack we show in Section 5, because the operating system is not trusted. Instead, the SGX interface can be changed to clear the BTB state on enclave entry and exit.

BunnyHop-Probe The cause of BunnyHop-Probe is resource contention within the set. One approach to prevent contention across hyperthreads is to statically partition the BTB rather than competitively sharing the BTB. Static partitioning only protects against attacks between hyperthreads. Therefore this mitigation would need to be deployed in combination with a time-sharing mitigation such as clearing the BTB between context switches.

Randomization has been proposed to protect against cache-based attacks [48, 60, 80]. Zhao et al. [92] propose a randomization-based approach for protecting the branch predictor.

Cryptography. Constant-time programming is a programming style that prohibits differences in observable program behavior based on program secrets [9, 13, 42]. Past works have demonstrated that relaxing constant-time requirements is risky [52, 63, 85]. Our BunnyHop-Evict attack demonstrates this once again.

10 Conclusion

In this work we explore the instruction prefetcher of Intel Core processors. We show that the prefetcher is directed by the branch predictor, enabling the BunnyHop technique, which transfers branch predictor state to cache state.

Our reverse engineering efforts show new structures in the Intel microarchitecture that allow us to perform new attacks that improve on existing work and overcome proposed defenses.

Acknowledgments

We thank the Intel technical team for the feedback on parts of this work.

This project has been supported by the Air Force Office of Scientific Research (AFOSR) under award number FA9550-20-1-0425; an ARC Discovery Early Career Researcher Award DE200101577; an ARC Discovery Project number DP210102670; CSIRO's Data61; the National Science Foundation under grant CNS-1954712; and gifts by AMD, Google, Intel, and Qualcomm;

Parts of this work were undertaken while Yuval Yarom was affiliated with Data61, CSIRO.

References

[1] 0xAde1a1de. AssemblyLine, 2022. URL https://github.com/0xAde1a1de/AssemblyLine.

[2] Andreas Abel and Jan Reineke. Measurement-based modeling of the cache replacement policy. In *RTAS*, pages 65–74, 2013. doi: 10.1109/RTAS.2013.6531080.

[3] Onur Acıiçmez. Yet another microarchitectural attack: exploiting I-cache. In *CSAW*, pages 11–18. ACM, 2007. doi: 10.1145/1314466.1314469.

[4] Onur Acıiçmez and Çetin Kaya Koç. Microarchitectural attacks and countermeasures. In *Cryptographic Engineering*, pages 475–504. Springer, 2009.

[5] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. Cryptology ePrint Archive, Report 2006/351, 2006. URL http://eprint.iacr.org/2006/351.

[6] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *CT-RSA*, pages 225–242, 2007. doi: 10.1007/11967668_15.

[7] Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In *CHES*, pages 110–124, 2010. doi: 10.1007/978-3-642-15031-9_8.

[8] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *IEEE SP*, pages 870–887, 2019. doi: 10.1109/SP.2019.00066.

[9] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *USENIX Security*, pages 53–70, 2016.

[10] *Indirect Branch Control Extension*. AMD Technology, October 2018.

[11] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. Branch history injection: On the effectiveness of hardware mitigations against cross-privilege Spectre-v2 attacks. In *USENIX Security*, pages 971–988, 2022. URL https://www.usenix.org/conference/usenixsecurity22/presentation/barberis.

[12] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. "ooh aah... just a little bit" : A small amount of side channel can go a long way. In *CHES*, pages 75–92, 2014. doi: 10.1007/978-3-662-44709-3_5.

[13] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In *Latincrypt*, pages 159–176, 2012.

[14] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: Exploiting speculative execution through port contention. In *CCS*, pages 785–800, 2019. doi: 10.1145/3319535.3363194.

[15] Robert Brotzman, Danfeng Zhang, Mahmut Taylan Kandemir, and Gang Tan. SpecSafe: detecting cache side channels in a speculative world. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–28, 2021. doi: 10.1145/3485506.

[16] Brad Calder and Dirk Grunwald. Next cache line and set prediction. In *ISCA*, pages 287–296, 1995.

[17] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *USENIX Security*, pages 249–266, 2019. URL https://www.usenix.org/conference/usenixsecurity19/presentation/canella.

[18] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten-Hwang Lai. SgxPectre: Stealing Intel secrets from SGX enclaves via speculative execution. In *EuroS&P*, pages 142–157, 2019.

[19] Wei Cheng, Jean-Luc Danger, Sylvain Guilley, Amina Bel Korchi, and Olivier Rioul. Cache-timing attack on the SEAL homomorphic encryption library. In *PROOFS*, Leuven, Belgium, 2022. URL https://hal.telecom-paris.fr/hal-03780506/document.

[20] Chitchanok Chuengsatiansup, Daniel Genkin, Yuval Yarom, and Zhiyuan Zhang. Side-channeling the Kalyna key expansion. In *CT-RSA*, pages 272–296, 2022. doi: 10.1007/978-3-030-95312-6_12.

[21] Thomas M. Conte, Kishore N. Menezes, Patrick M. Mills, and Burzin A. Patel. Optimization of instruction fetch mechanisms for high issue rates. In *ISCA*, pages 333–344, 1995.

[22] Intel Corp. Guidelines for mitigating timing side channels against cryptographic implementations. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/mitigate-timing-side-channel-crypto-implementation.html, 2022.

[23] Shuwen Deng, Bowen Huang, and Jakub Szefer. Leaky frontends: Security vulnerabilities in processor frontends. In *HPCA*, pages 53–66, 2022. doi: 10.1109/HPCA53966.2022.00013.

[24] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *MICRO*, pages 1–13, 2016. doi: 10.1109/MICRO.2016.7783743.

[25] Dmitry Evtyushkin, Ryan Riley, Nael B. Abu-Ghazaleh, and Dmitry Ponomarev. BranchScope: A new side-channel attack on directional branch predictor. In *ASPLOS*, pages 693–707, 2018. doi: 10.1145/3173162.3173204.

[26] Shuqin Fan, Wenbo Wang, and Qingfeng Cheng. Attacking OpenSSL implementation of ECDSA with a few signatures. In *CCS*, pages 1505–1515, 2016. doi: 10.1145/2976749.2978400.

[27] Michael Ferdman, Cansu Kaynak, and Babak Falsafi. Proactive instruction fetch. In *MICRO*, pages 152–162, 2011.

[28] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptogr. Eng.*, 8(1):1–27, 2018. doi: 10.1007/s13389-016-0141-6.

[29] Daniel Genkin, Luke Valenta, and Yuval Yarom. May the fourth be with you: A microarchitectural side channel attack on several real-world applications of Curve25519. In *CCS*, pages 845–858, 2017. doi: 10.1145/3133956.3134029.

[30] Google. Retpoline: a software construct for preventing branch-target-injection, 2018. URL https://support.google.com/faqs/answer/7625886.

[31] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *USENIX Security*, pages 955–972, 2018. URL https://www.usenix.org/conference/usenixsecurity18/presentation/gras.

[32] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security*, pages 897–912, 2015. URL https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/gruss.

[33] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games - bringing access-based cache attacks on AES to practice. In *IEEE SP*, pages 490–505, 2011. doi: 10.1109/SP.2011.22.

[34] Berk Gülmezoglu, Andreas Zankl, M. Caner Tol, Saad Islam, Thomas Eisenbarth, and Berk Sunar. Undermining user privacy on mobile devices using AI. In *AsiaCCS*, pages 214–227, 2019. doi: 10.1145/3321705.3329804.

[35] Yanan Guo, Andrew Zigerelli, Youtao Zhang, and Jun Yang. Adversarial prefetch: New cross-core cache side channel attacks. In *IEEE SP*, pages 1458–1473, 2022. doi: 10.1109/SP46214.2022.9833692.

[36] Norman Hardy. The confused deputy (or why capabilities might have been invented). *ACM SIGOPS Oper. Syst. Rev.*, 22(4):36–38, 1988. doi: 10.1145/54289.871709.

[37] Jan Hoogerbrugge. Cost-efficient branch target buffers. In *Euro-Par*, pages 950–959, 2000. doi: 10.1007/3-540-44520-X_134.

[38] Tianlin Huo, Xiaoni Meng, Wenhao Wang, Chunliang Hao, Pei Zhao, Jian Zhai, and Mingshu Li. Bluethunder: A 2-level directional predictor based side-channel attack against SGX. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(1):321–347, 2020. doi: 10.13154/tches.v2020.i1.321-347.

[39] Intel. Deep dive: Retpoline: A branch target injection mitigation., 2018.

[40] Intel Corporation. Speculative execution side channel mitigations. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/speculative-execution-side-channel-mitigations.html, May 2018.

[41] *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation, February 2022.

[42] Jan Jancar, Marcel Fourné, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. "they're not that hard to mitigate": What cryptographic library developers think about timing attacks. In *IEEE SP*, pages 632–649, 2022.

[43] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE SP*, pages 1–19, 2019. doi: 10.1109/SP.2019.00002.

[44] Rakesh Kumar, Cheng-Chieh Huang, Boris Grot, and Vijay Nagarajan. Boomerang: A metadata-free architecture for control flow delivery. In *HPCA*, pages 493–504, 2017.

[45] 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 *USENIX Security*, pages 557–574, 2017.

[46] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security*, 2018. URL https://www.usenix.org/conference/usenixsecurity18/presentation/lipp.

[47] Moritz Lipp, Andreas Kogler, David F. Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. PLATYPUS: software-based power side-channel attacks on x86. In *IEEE SP*, pages 355–371, 2021. doi: 10.1109/SP40001.2021.00063.

[48] Fangfei Liu and Ruby B. Lee. Random fill cache architecture. In *MICRO*, pages 203–215, 2014. doi: 10.1109/MICRO.2014.28.

[49] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *IEEE SP*, pages 605–622, 2015. doi: 10.1109/SP.2015.43.

[50] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. A survey of microarchitectural side-channel vulnerabilities, attacks, and defenses in cryptography. *ACM Comput. Surv.*, 54(6):122:1–122:37, 2021. doi: 10.1145/3456629.

[51] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX amplifies the power of cache attacks. In *CHES*, pages 69–90, 2017. doi: 10.1007/978-3-319-66787-4_4.

[52] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. MemJam: A false dependency attack against constant-time crypto implementations in SGX. In *CT-RSA*, pages 21–44, 2018. doi: 10.1007/978-3-319-76953-0_2.

[53] Michael Neve and Jean-Pierre Seifert. Advances on access-driven cache attacks on AES. In *SAC*, pages 147–162, 2006. doi: 10.1007/978-3-540-74462-7_11.

[54] José Luiz Negreira Castro de Oliviera and Diego Leonel Cadette Dutra. Reverse branch target buffer poisoning. Relatório Technico ES-783/22, Universidade Federal do Rio de Janeiro, September 2022. URL https://cos.ufrj.br/index.php/pt-BR/publicacoes-pesquisa/details/15/3061.

[55] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *CT-RSA*, pages 1–20, 2006. doi: 10.1007/11605805_1.

[56] Riccardo Paccagnella, Licheng Luo, and Christopher W. Fletcher. Lord of the ring(s): Side channel attacks on the CPU on-chip ring interconnect are practical. In *USENIX Security Symposium*, pages 645–662, 2021. URL https://www.usenix.org/conference/usenixsecurity21/presentation/paccagnella.

[57] Colin Percival. Cache missing for fun and profit. In *BSDCon 2005*, Ottawa, CA, 2005. URL https://www.daemonology.net/papers/htt.pdf.

[58] Jop van de Pol, Nigel P. Smart, and Yuval Yarom. Just a little bit more. In *CT-RSA*, pages 3–21, 2015. doi: 10.1007/978-3-319-16715-2_1.

[59] Ivan Puddu, Moritz Schneider, Miro Haller, and Srdjan Capkun. Frontal attack: Leaking control-flow in SGX via the CPU frontend. In *USENIX Security*, pages 663–680, 2021. URL https://www.usenix.org/conference/usenixsecurity21/presentation/puddu.

[60] Moinuddin K. Qureshi. New attacks and defense for encrypted-address cache. In *ISCA*, pages 360–371, 2019. doi: 10.1145/3307650.3322246.

[61] Glenn Reinman, Brad Calder, and Austin. Fetch directed instruction prefetching. In *MICRO*, pages 16–27, 1999. doi: 10.1109/MICRO.1999.809439.

[62] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M. Tullsen, and Ashish Venkat. I see dead µops: Leaking secrets via Intel/AMD micro-op caches. In *ISCA*, pages 361–374, 2021. doi: 10.1109/ISCA52012.2021.00036.

[63] Eyal Ronen, Kenneth G. Paterson, and Adi Shamir. Pseudo constant time implementations of TLS are only pseudo secure. In *CCS*, pages 1397–1414, 2018.

[64] Eyal Ronen, Robert Gillham, Daniel Genkin, Adi Shamir, David Wong, and Yuval Yarom. The 9 lives of Bleichenbacher's cat: New cache attacks on TLS implementations. In *IEEE SP*, pages 435–452, 2019. doi: 10.1109/SP.2019.00062.

[65] Aria Shahverdi, Mahammad Shirinov, and Dana Dachman-Soled. Database reconstruction from noisy volumes: A cache side-channel attack on SQLite. In *USENIX Security*, pages 1019–1035, 2021. URL https://www.usenix.org/conference/usenixsecurity21/presentation/shahverdi.

[66] Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *DSN Workshops*, pages 194–199, 2011.

[67] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust website fingerprinting through the cache occupancy channel. In *USENIX Security*, pages 639–656, 2019. URL https://www.usenix.org/conference/usenixsecurity19/presentation/shusterman.

[68] Anatoly Shusterman, Ayush Agarwal, Sioli O'Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. Prime+Probe 1, JavaScript 0: Overcoming browser-based side-channel defenses. In *USENIX Security*, pages 2863–2880, 2021. URL https://www.usenix.org/conference/usenixsecurity21/presentation/shusterman.

[69] Florian Sieck, Sebastian Berndt, Jan Wichelmann, and Thomas Eisenbarth. Util:Lookup: Exploiting key decoding in cryptographic libraries. In *CCS*, pages 2456–2473, 2021. doi: 10.1145/3460120.3484783.

[70] Viji Srinivasan, Edward S. Davidson, Gary S. Tyson, Mark J. Charney, and Thomas R. Puzak. Branch history guided instruction prefetching. In *HPCA*, pages 291–300, 2001.

[71] Jakub Szefer. Survey of microarchitectural side and covert channels, attacks, and defenses. *J. Hardw. Syst. Secur.*, 3(3):219–234, 2019. doi: 10.1007/s41635-018-0046-1.

[72] Mohammadkazem Taram, Xida Ren, Ashish Venkat, and Dean Tullsen. SecSMT: Securing SMT processors against contention-based covert channels. In *USENIX Security*, 2022.

[73] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. Meltdown-Prime and SpectrePrime: Automatically-synthesized attacks exploiting invalidation-based coherence protocols. CORR arXiv 1802.03802 http://arxiv.org/abs/1802.03802, 2018.

[74] Yukiyasu Tsunoo, Etsuko Tsujihara, Kazuhiko Minematsu, and Hiroshi Hiyauchi. Cryptanalysis of block ciphers implemented on computers with cache. In *International Symposium on Information Theory and Its Applications*, pages 803–806, 2002.

[75] Vladimir Uzelac and Aleksandar Milenkovic. Experiment flows and microbenchmarks for reverse engineering of branch predictor structures. In *ISPASS*, pages 207–217, 2009. doi: 10.1109/ISPASS.2009.4919652.

[76] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A practical attack framework for precise enclave execution control. In *SysTEX*, pages 6–11, 2017.

[77] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic. In *CCS*, pages 178–195, 2018. doi: 10.1145/3243734.3243822.

[78] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: hijacking transient execution through microarchitectural load value injection. In *IEEE SP*, pages 54–72, 2020. doi: 10.1109/SP40000.2020.00089.

[79] Jose Rodrigo Sanchez Vicarte, Michael Flanders, Riccardo Paccagnella, Grant Garrett-Grossman, Adam Morrison, Christopher W. Fletcher, and David Kohlbrenner. Augury: Using data memory-dependent prefetchers to leak data at rest. In *IEEE SP*, pages 1491–1505, 2022. doi: 10.1109/SP46214.2022.9833570.

[80] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. ScatterCache: Thwarting cache attacks via cache set randomization. In *USENIX Security*, pages 675–692, 2019. URL https://www.usenix.org/conference/usenixsecurity19/presentation/werner.

[81] Johannes Wikner and Kaveh Razavi. RETBLEED: arbitrary speculative code execution with return instructions. In *USENIX*, pages 3825–3842, 2022.

[82] Wenjie Xiong and Jakub Szefer. Survey of transient execution attacks and their mitigations. *ACM Comput. Surv.*, 54(3):54:1–54:36, 2021. doi: 10.1145/3442479.

[83] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher W. Fletcher, Roy H. Campbell, and Josep Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *IEEE SP*, pages 888–904, 2019. doi: 10.1109/SP.2019.00004.

[84] Yuval Yarom and Katrina Falkner. Flush+Reload: a high resolution, low noise, L3 cache side-channel attack. In *USENIX Security*, pages 719–732, 2014. URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.

[85] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: A timing attack on OpenSSL constant time RSA. In *CHES*, pages 346–367, 2016. doi: 10.1007/978-3-662-53140-2_17.

[86] Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. COLORIS: A dynamic cache partitioning system using page coloring. In *PACT*, pages 381–392, 2014.

[87] Rui Zhang, Michael D. Bond, and Yinqian Zhang. Cape: compiler-aided program transformation for HTM-based cache side-channel defense. In *CC*, pages 181–193, 2022. doi: 10.1145/3497776.3517778.

[88] Tao Zhang, Kenneth Koltermann, and Dmitry Evtyushkin. Exploring branch predictors for constructing transient execution Trojans. In *ASPLOS*, pages 667–682, 2020. doi: 10.1145/3373376.3378526.

[89] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. In *EuroSys*, pages 89–102, 2009.

[90] Yi Zhang, Steve Haga, and Rajeev Barua. Execution history guided instruction prefetching. In *ICS*, pages 199–208, 2002.

[91] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *CCS*, pages 305–316, 2012. doi: 10.1145/2382196.2382230.

[92] Lutan Zhao, Peinan Li, Rui Hou, Michael C. Huang, Xuehai Qian, Lixin Zhang, and Dan Meng. HyBP: Hybrid isolation-randomization secure branch predictor. In *HPCA*, pages 346–359, 2022.

A Compare BunnyHop with Phantom JMPs

Wikner and Razavi [81] reports that on some AMD processors an arbitrary instruction can trigger speculative execution, which is called Phantom JMP. To force the misprediction of an arbitrary instruction, the branch history of the instruction needs to match the branch history of an indirect branch. Unlike other spectre-BTB attacks [11, 43], the speculative window of Phantom JMP is very short which allows only one instruction to be speculatively executed. The root cause of Phantom JMP is suspected to be the instruction prefetcher on AMD processors.

In this paper, we reverse engineer the instruction prefetcher on modern Intel processors in Section 3 and we show that arbitrary instruction can be mispredicted to prefetch memory blocks from the branch target. To compare with Phantom JMP, we conduct experiments in Section 3 on AMD Ryzen 9 5950X running Ubuntu 22.04 TLS with micro-code 0x0a201016. We firstly verify that the prefetch depth is 15 memory blocks following the `RET` instruction. Then we repeat the experiment in Section 3.5 to investigate the prefetching depth with temporal restrictions. The result is shown in Figure 7. On the tested AMD machine, we find that the prefetch depth is not affected by the BTB content which is contrary to the behavior in Figure 2. Furthermore, we observe that when the hyperthread is active, the sibling thread constantly prefetches five memory blocks ahead.

Figure 7 shows that the instruction prefetcher is not affected by the BTB content. Note that Phantom JMP requires

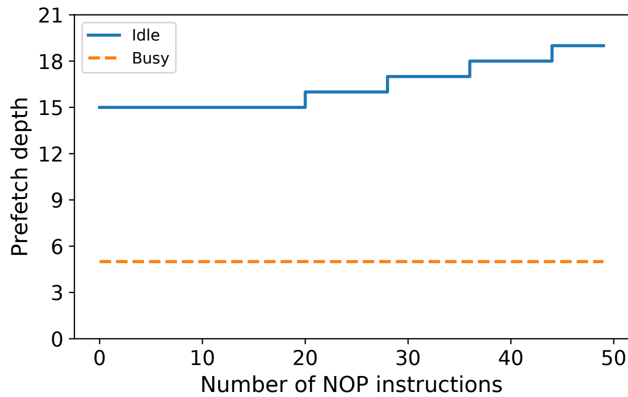

Figure 7: The temporal restriction does not affect the prefetching depth on AMD Ryzen 9 5950X.

matching the branch history of an arbitrary instruction and an indirect branch. We hypothesize that the prediction of instruction prefetcher of AMD processor has multiple modes. Firstly, if the fetched instruction matches the branch history of an indirect branch, the instruction prefetcher starts prefetching at the target address. At the same time, the back-end engine starts speculative execution. In the scenario that there is no match in the branch history buffer, the instruction prefetcher simply prefetches 15 memory blocks ahead. We leave the task to investigate the instruction prefetcher on AMD machine as an future work.