OCCUPY+PROBE: Cross-Privilege Branch Target Buffer Side-Channel Attacks at Instruction Granularity

Kaiyuan Rong*‡, Junqi Fang*‡, Haixia Wang††, Dapeng Ju*‡, and Dongsheng Wang*‡
*Department of Computer Science and Technology, Tsinghua University
†Beijing National Research Center for Information Science and Technology, Tsinghua University
‡Zhongguancun Laboratory
{rky22, fangjq24}@mails.tsinghua.edu.cn, {hx-wang, judapeng, wds}@tsinghua.edu.cn

***Abstract*—In recent years, the Branch Target Buffer (BTB) has raised significant concerns in system security research. As this component is logically or physically shared in certain attack scenarios, it is abused by adversaries to construct side-channels that leak sensitive branch information of victim processes. However, existing BTB side-channel attacks either fail to leak kernel control-flow information from user mode due to the cross-privilege isolation mechanism, or suffer from limited spatial resolution in branch monitoring.**

In this paper, we propose OCCUPY+PROBE, a novel eviction-based BTB side-channel attack that bridges these gaps by successfully exposing kernel control-flow behaviors directly from user mode. Our approach begins with an in-depth reverse engineering of the offset-related BTB update mechanism on Intel processors, and reveals that *BTB entries created in user mode can be directly replaced by kernel-mode entries, irrespective of the underlying replacement policy and the hardware isolation*, which forms the foundation of OCCUPY+PROBE. In contrast to existing BTB side-channel attacks, OCCUPY+PROBE eliminates the need for entry sharing between the attacker and the victim. Moreover, it achieves instruction-level granularity in branch monitoring, surpassing the spatial resolution of existing eviction-based BTB side-channels.

We experimentally demonstrate that OCCUPY+PROBE can leak control-flow information across privilege boundaries with high spatial resolution on various Intel processors. Furthermore, we validate the practical effectiveness of OCCUPY+PROBE through a detailed case study targeting the Linux Kernel Crypto API, showcasing its potential to compromise critical kernel operations. Additionally, compared to prior eviction-based BTB side-channels, OCCUPY+PROBE demonstrates a unique capability to extract tag values of kernel branches, which can be exploited to break KASLR.

†Corresponding author.

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.230925
www.ndss-symposium.org

I. INTRODUCTION

In the past decade, microarchitectural attacks [7], [11], [15], [16], [21], [24], [26]–[28], [39], [47], [48], [50], [52], [54] targeting the Branch Prediction Unit (BPU) have emerged in rapid succession. These works either analyze publicly known microarchitectural components to identify security vulnerabilities and construct attacks, or perform reverse engineering on undocumented components to uncover their internal mechanisms and microarchitectural behaviors, thereby exposing potential security risks. Microarchitectural threats stemming from branch prediction logic can be broadly classified into side-channel attacks [1]–[3], [15], [16], [21], [27], [50], [52], [54] and transient execution attacks [7], [11], [24], [26], [47], [48], [53], both of which present serious risks to processor security.

The Branch Target Buffer (BTB) is one of the most extensively studied components within the BPU, and its observable behavior caused by branch mispredictions can be exploited to construct side-channel attacks [1]–[3], [15], [27], [52], [54]. These attacks demonstrate the capability to leak control-flow information across processes [1]–[3], [15], [52], [54], across privilege boundaries [15], [54], and even across Intel SGX enclaves [27], [52], [54]. Furthermore, several studies [27], [41], [53], [54] conduct in-depth reverse-engineering of BTB structures, revealing detailed design information of BTBs in modern processors. These findings both support and enhance the effectiveness of BTB-based side-channel attacks.

However, existing BTB side-channel attacks face limitations when applied to cross-privilege scenarios on Intel processors. Due to the hardware isolation mechanism introduced starting with the 11th-generation Intel Core processors (cf. Section III-A), attacks that rely on accessing victim BTB entries [15], [27], [52], [54] can no longer leak the execution of kernel branches from user mode. Others exploiting BTB evictions to leak the control-flow [1]–[3], [54] suffer from limited spatial resolution, as they only allow observation at the granularity of BTB sets. That is, they can infer the presence of branch execution within a set, but are unable to attribute

it to a specific branch instruction. This lack of instruction-level granularity introduces ambiguity when multiple branch instructions map to the same BTB set, making it infeasible to accurately determine which specific branch has been executed. Consequently, such attacks cannot reliably infer fine-grained control-flow information, which may be critical for recovering sensitive data or understanding victim behavior. These limitations pose significant challenges to constructing an effective cross-privilege BTB side-channel attack.

In this paper, we propose OCCUPY+PROBE, a novel cross-privilege and eviction-based BTB side-channel attack that successfully overcomes the challenges mentioned earlier. Our experiments demonstrate that OCCUPY+PROBE can leak kernel branch execution outcomes from user mode, even on processors equipped with hardware isolation that prevents user programs from accessing kernel BTB entries. Furthermore, OCCUPY+PROBE achieves instruction-level granularity, matching the theoretical maximum spatial resolution of BTB side-channel attacks. In addition, unlike existing eviction-based attacks, OCCUPY+PROBE can extract the tag values of BTB entries, enabling a reliable KASLR break on Intel Core i7-11700K processors.

To construct OCCUPY+PROBE, we conduct an in-depth reverse engineering of the offset-related BTB update mechanism, focusing on how the BTB is updated based on the offset values of its entries, a previously unexplored behavior in the literature. Before conducting our experiments, we first replicate the prior work [54] to obtain fundamental details of the BTB structures on our target processors. Table I summarizes key characteristics of the BTB across several Intel Core generations, from the 9th to the 14th, which serve as the foundation for our reverse engineering analysis. We then delve into the offset-related BTB update mechanism and find that the BTB uses the address of the last byte within branch instructions for indexing, and employs four distinct update mechanisms, each dependent on the offset values of BTB entries. Furthermore, we analyze the update behavior across different execution domains and uncover that entries created in user mode can be directly replaced by those from kernel mode, regardless of the underlying replacement policy or hardware isolation. This property is the core insight that enables OCCUPY+PROBE.

To demonstrate the practical impact of OCCUPY+PROBE, we launch an end-to-end attack in a real-world scenario, targeting the leakage of a private RSA key used during decryption within the Linux Kernel Crypto API [31]. To achieve this, we design an attack model that repeatedly interrupts the decryption process, allowing us to observe the execution outcomes of secret-dependent branches across iterations and thereby enable key recovery. Our experimental results show that the recovered secret bits achieve an accuracy of 98%. Finally, we discuss potential defenses against OCCUPY+PROBE.

Contributions. In summary, this work makes the following contributions:

- We conduct an in-depth reverse engineering of the offset-related BTB update mechanism on Intel processors, uncovering four distinct update mechanisms and their behaviors across different execution domains.
- We propose OCCUPY+PROBE, a novel cross-privilege, eviction-based BTB side-channel attack that succeeds on BTB-hardened processors and achieves instruction-level spatial resolution.
- We demonstrate the practicality of OCCUPY+PROBE through an end-to-end attack on the RSA algorithm implemented in the Linux Kernel Crypto API, successfully recovering the private key with 98% accuracy.

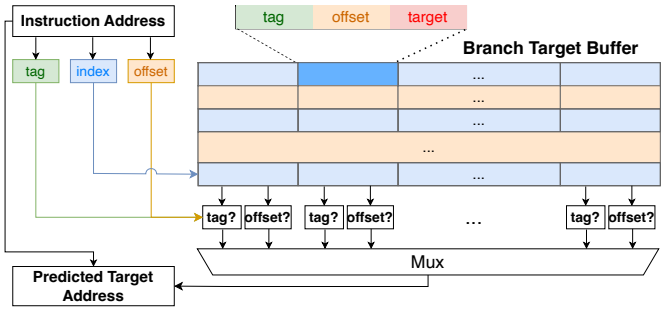

Fig. 1. The BTB structure and branch prediction logic.

Responsible Disclosure. We reported our findings to Intel on April 18, 2025. Intel confirmed our findings, and stated that the underlying attack scenario can be addressed by their existing side-channel guidance[1] and no additional mitigations are necessary.

Availability. The experiments and proof-of-concept implementations of this paper are available at https://github.com/CPU-Security/Occupy_Probe.

II. BACKGROUND

A. Branch Target Buffer

BTB Structure. In modern processors, the Branch Target Buffer (BTB) [20], [41], [54] is a key component of the Branch Prediction Unit (BPU), recording executed branch instructions to assist in predicting future control flow. Figure 1 illustrates the basic structure of a typical set-associative BTB. The virtual address of a branch instruction is used to generate the BTB index, tag, and offset. The index determines the BTB set, while the tag, offset, lower bits of the target address, and other branch-related information are stored in the BTB entry. During the prediction phase, the tag and offset of the instruction address are compared with those of all entries in the BTB set identified by the index, to determine whether any entry matches the instruction. The matching entry is used to predict the target address by concatenating the target recorded in the entry with the higher bits of the branch address. Additionally, NIGHTVISION [52] reveals that the BTB selects the entry with the smallest offset among those whose index and tag match and whose offset is no less than the PC offset.

[1] https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/security-best-practices-side-channel-resistance.html

TABLE I
BTB DETAILS OF VARIOUS INTEL CORE CPUS.

Model	i7-9700	i9-10850K	i7-11700K	i7-12700K (P-core) i9-13900K (P-core) i9-14900K (P-core)	i7-12700K (E-core) i9-13900K (E-core) i9-14900K (E-core)
μarch	Coffee Lake	Comet Lake	Rocket Lake	Raptor (Golden) Cove	Gracemont
Index	[13:5]	[13:5]	[13:5]	[14:5]	[14:5]
Tag	[29:22] $\oplus$ [21:14]	[29:22] $\oplus$ [21:14]	[33:24] $\oplus$ [23:14]	[23:15]	[24:15]
Offset	[4:0]	[4:0]	[4:0]	[4:0]	[4:0]
Associativity	4+4 (Short+Long)[1]	4+4	6+4	12	5
Target	10 / 32 LSB[2]	10 / 32 LSB	12 / 32 LSB	32 LSB	32 LSB

[1] Short+Long means the number of short ways and long ways.
[2] LSB means the Least Significant Bits.

Explored BTB Mechanisms. Prior research [15], [27], [39], [48], [52]–[54] has revealed various characteristics of the BTB, which we classify into two categories: *prediction* and *update*. Regarding prediction behaviors, branches colliding in the BTB can share the same branch prediction. Specifically, the BTB uses a subset of instruction address bits for indexing. Therefore, branch instructions with the same lower address bits can use each other's BTB entries for branch prediction. Moreover, there are observable effects for mispredictions, for example, through timing measurements [15], covert channels [39], [48], instruction prefetching [54] (or speculative fetch [48]), and some hardware features [27].

For update behaviors, when inserting a new entry, if no entry in the corresponding BTB set has the same tag as the new entry, a candidate entry is evicted based on the replacement policy to insert the new one [54]. However, in cases where an entry with a matching tag already exists, the BTB's update strategy remains unexplored in the academic literature. Our findings reveal that the BTB employs different update mechanisms based on the offsets between existing and new branches, with these mechanisms varying across execution domains, as discussed in Section IV. We refer to this mechanism as the *offset-related* BTB update mechanism.

B. BTB Side-Channel Attacks

Many studies [1]–[3], [15], [27], [52], [54] on the BTB have leveraged their findings to construct BTB-based side-channel attacks. One purpose of these attacks is to check whether a specific victim branch is executed or taken. Prior works motivated by this reason can be categorized into *access-based* and *eviction-based* attacks, analogous in principle to the FLUSH+RELOAD [49] and PRIME+PROBE [32], [38] cache side-channel attacks, respectively.

Access-based. In access-based attacks [15], [27], [52], [54], the attacker executes their own branch instruction colliding with the victim branch in the BTB, and observes the prediction result to determine whether a victim entry was created after the victim execution. For instance, Branch Shadowing [27] uses a shadow branch, whose least significant bits match those of the enclave branch to probe the BTB state after the enclave executes, thereby inferring the control flow of the enclave. BunnyHop-Reload [54] employs a similar strategy but replaces timing measurements with instruction prefetch to obtain the BTB state. JumpOverASLR [15] leverages BTB collisions to infer whether the guessed address matches the address of a specific victim branch, enabling breaking (K)ASLR. [2]

Eviction-based. Another type of attack [1]–[3], [54] is eviction-based, which abuses BTB eviction to infer victim branch execution. In particular, the attacker first primes an entire BTB set, and then the executed victim branch can evict a primed entry of the attacker, which can be observed by the attacker if those branches are mapped to the same BTB set. Onur Acıiççmez et al. [1]–[3] proposed a technique that primes an entire BTB set and measures execution time to determine whether a victim branch mapped to the same set has evicted a primed branch. BunnyHop-Probe [54] improves upon this approach by narrowing the probing scope to a single candidate branch within the prime set, which is strategically chosen for eviction based on the replacement policy, allowing the attacker to detect victim branch execution more efficiently. We generally refer to this type of attack as BTB PRIME+PROBE attack.

C. Instruction Prefetch

Prior research has demonstrated that the memory line of a BTB-predicted target is loaded into the instruction cache (I-Cache), either via instruction prefetch [54] or speculative fetch [48]. Based on this insight, the BTB state can be transferred into the I-Cache state, allowing it to be probed using the FLUSH+RELOAD technique [49]. As illustrated in Figure 2, `Train` and `Probe` collide in the BTB (e.g. sharing lower address bits). After invoking `Train`, which inserts a corresponding entry in the BTB, the execution of `Probe` will trigger the prefetch of `Predict` into I-Cache, as `Predict` is the predicted target address generated by the BTB. In this paper, we utilize this mechanism to extract the BTB prediction result. Unlike traditional approaches that rely on timing measurements or other hardware features to observe branch mispredictions, this method provides a direct and efficient way to retrieve predicted targets and verify pre-trained BTB entries.

[2]Some studies [18], [54], [55] classify JumpOverASLR as an eviction-based attack. In this work, we conduct a thorough analysis based on our findings and ultimately conclude that it is an access-based attack, as discussed in Section IV-E.

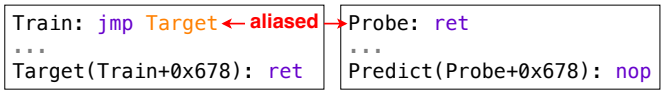

Fig. 2. Code that contains two branches colliding in the BTB. The `Probe` branch can use the prediction of `Train`, causing `Predict` being prefetched into I-Cache.

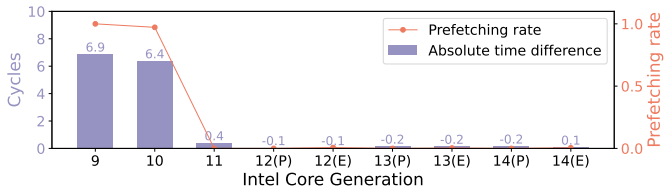

Fig. 3. Cross-privilege BTB sharing across Intel Core generations. Bars show user branch timing differences with and without kernel branch pre-training; values above indicate exact differences.

III. Overview of Occupy+Probe

A. Motivation

As described in Section II-B, existing BTB side-channel attacks can be divided into access-based and eviction-based attacks. However, the two types of attacks have obvious drawbacks under the cross-privilege scenario. Access-based attacks heavily rely on the property that the BTB is logically shared between the attacker and the victim, meaning that the attacker is capable to use BTB entries created by the victim for branch prediction. Unfortunately, our experimental results indicate that starting from the 11th generation of Intel Core processors, user programs are prevented from utilizing kernel BTB entries for branch prediction.

We confirm this finding with an experiment performed on the processors listed in Table I. In this experiment, we construct a user branch and a kernel branch that collide in the BTB, and examine the branch prediction behavior using timing measurements [15] and instruction prefetching [54], respectively. For the timing test, the jump lengths of the two branches are identical. The experiment measures the execution time of the user branch after executing the kernel branch and without executing the kernel branch, respectively, and the time difference is presented in the bar chart in Figure 3. For the prefetching test, we measure, after executing the kernel branch, whether the execution of the user branch would trigger the prefetching of the predicted address. The prefetching rate is shown as a line chart in Figure 3.

According to the results, on 9th and 10th generation processors, the execution time of the user branch is approximately 7 cycles faster after executing the kernel branch, and prefetching is observed. However, this behavior is not observed on processors from the 11th generation onward. This proves that access-based attacks can no longer leak the execution of kernel branches from user mode.

For eviction-based attacks, the spatial resolution is relatively low, as it can only determine whether a victim branch has been executed within a BTB set. It cannot reveal finer details, such as which specific branch instruction within a 32-byte-aligned code block was executed. Moreover, these attacks are unable to infer the tag value of the victim branch since eviction is triggered solely based on index matches. Eviction-based attacks are also relatively inefficient, as they require executing a number of branches equal to the BTB's associativity to induce eviction.

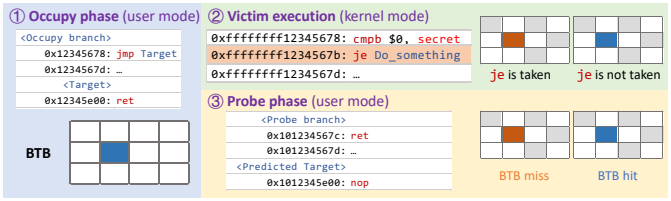

Fig. 4. The workflow of Occupy+Probe.

To address the above limitations of existing BTB side-channel attacks, we propose a novel BTB side-channel that successfully leaks kernel branch information from user mode at instruction granularity. The workflow of this attack is introduced in Section III-C.

B. Threat Model

We consider an attack scenario in which the attacker operates in user mode while the victim runs in kernel mode. Both attacker and victim threads execute on the same logical core. On 12th–14th generation Intel Core processors, the two threads specifically run on a performance core, a prerequisite for our proposed attack. The attacker can execute arbitrary unprivileged code on the system, with all hardware mitigations against Spectre-v2 [7], [24] (e.g., Indirect Branch Restricted Speculation (IBRS) [14] and Single Thread Indirect Branch Predictors (STIBP) [14]) enabled on the processor. In addition, the attacker can execute the victim code multiple times, for example, via system calls. Similar to previous BTB side-channel attacks, we assume the victim program contains a branch dependent on a secret, with the virtual address of this branch known to the attacker. The objective of the attacker is to determine whether the branch is taken, thereby inferring the secret value.

C. Workflow

In this paper, we propose Occupy+Probe, a cross-privilege and eviction-based BTB side-channel attack based on our findings, which are detailed in Section IV. Figure 4 shows the workflow of Occupy+Probe, containing three main steps:

1) **Occupy Phase:** The attacker constructs an *occupy* branch whose address is carefully chosen based on the target victim branch's address. By executing this single branch, the attacker inserts a corresponding entry into the BTB.
2) **Victim Execution:** The victim process runs, executing the target branch, which may alter the state of the BTB entry established during the occupy phase.
3) **Probe Phase:** The attacker then probes the previously inserted BTB entry using an aliased branch to infer the

execution of the victim branch: (a) If the victim branch is taken, it directly replaces the *occupy* branch, bypassing the usual replacement policy and hardware isolation. This replacement behavior is explained in detail in Section IV. (b) If the victim branch is not taken, no replacement occurs, and the *occupy* branch remains, allowing the attacker to hit it.

In summary, if the victim branch is secret-dependent, its execution outcome is leaked via OCCUPY+PROBE, allowing the secret to be recovered by the attacker. In Section IV, we investigate the offset-related BTB update mechanism to support OCCUPY+PROBE, and then evaluate OCCUPY+PROBE in the subsequent sections.

IV. OFFSET-RELATED BTB UPDATE MECHANISM

In this section, we present our investigation into the offset-related BTB update mechanism. More specifically, we aim to address the following question:

> *When the BTB already contains an entry with the same index and tag as a newly executed branch, how does the BTB update its contents based on the offsets of the two entries?*

To answer this question, we first determine the address of which byte in the branch instruction the BTB uses for indexing (Section IV-A). We then evaluate the validity of the two branches (*i.e.*, the existing branch and the newly executed branch) in the BTB after the offset-related update mechanism by testing different offset pairs (Section IV-B). Next, in Section IV-C, we measure the number of entries affected by the offset-related update mechanism in the same BTB set to infer its underlying update behavior. We also investigate the offset-related update mechanism across different execution domains (Section IV-D). Finally, in Section IV-E, we revisit and analyze the root cause of JumpOverASLR [15] based on our findings, as this sheds light on the fundamental differences between their approach and our proposed attack.

HW/SW configurations. All experiments described in Section (IV-A)–(IV-D) are conducted on Intel processors listed in Table I, running Ubuntu 20.04 or 22.04. All hardware mitigations against Spectre-v2 [7], [24] are enabled on these processors.

A. BTB Indexing Address

The initial question we aim to address is the specific byte within a branch instruction from which the offset field is sourced. Previous studies [15], [27], [28], [41], [53], [54] have established that the BTB indexing function uses the instruction address of a branch as input, and the offset field of a BTB entry records the least significant 5 bits of the address (see Table I). However, most branch instructions comprise multiple bytes, each of which has its own memory address. Therefore, we need to investigate which specific address is used for indexing and being recorded in the offset field.

Experiment setup. To this end, we conduct an experiment, as illustrated in Figure 5. We create a `Train` branch of a direct jump instruction (5-byte length), and position it across a 32-byte memory boundary by inserting x `nops` before the `jmp` instruction (i.e. $28 \leq x \leq 31$), in which case addresses of different bytes can be mapped to two different BTB sets.

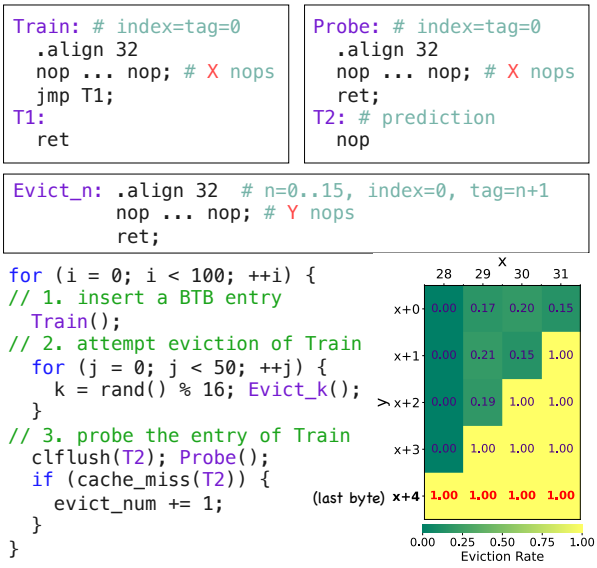

Fig. 5. The experimental design and result for exploring the BTB indexing address.

Next, we create 16 `Evict` branches that collide with a specific byte of the `Train` branch in one BTB set, based on the value y ($x \leq y \leq x+4$). All `Evict` branches are return instructions, as a return instruction is only 1 byte in length, allowing us to determine which BTB set they are mapped to. The key point of this experiment is that when these `Evict` branches and the `Train` branch are mapped to the same set, these branches can successfully evict the `Train` branch after their executions, helping us determine whether the specific byte in the `Train` branch is a possible indexing byte.

In each run, we insert the `Train` entry, execute all `Evict` branches multiple times, and finally probe with an aliasing branch to check if `Train` was evicted. This process is repeated 100 times to compute the eviction rate.

Result. The result shows that for `Train` branches crossing a 32-byte boundary, only the eviction set corresponding to the *last* byte of the `Train` branch always evicts it. This result indicates that the BTB uses the *last* byte of each branch instruction for indexing, which is consistent with the Intel patent [20], and the offset field records the least significant 5 bits of the address of this byte.

> **Observation 1.** The BTB is indexed with the address of the last byte of each branch instruction.

B. Observing Valid State of Entries after BTB Update

As mentioned earlier, when the index and tag of a newly executed branch match those of an existing entry in the BTB, the BTB is updated based on their offsets. We refer to the

```
1  void *Texist;      // Addr: (0x1 << 36) + 0xc - 4
2                     // jump to ((0x1 << 36) + 0x600)
3  void *Tnew;        // Addr: (0x2 << 36) + 0xf - 4
4                     // jump to ((0x2 << 36) + 0xc00)
5  void *Pexist;      // Addr: (0x3 << 36) + 0xc
6  void *Pnew;        // Addr: (0x3 << 36) + 0xf
7  void *pred_exist;  // Addr: (0x3 << 36) + 0x600
8  void *pred_new;    // Addr: (0x3 << 36) + 0xc00
9  inline void mfence_delay(int loop_num) {
10   asm volatile("mfence");
11   for (int z = 0; z < loop_num; ++z) {
12     asm volatile("nop");
13   }
14 }
15 double get_hit_rate(void *probe, void *probe_addr) {
16   int count = 0;
17   for (int i = 0; i < TOTAL; ++i) {
18     flush_BTB();
19     clflush(probe_addr);
20     mfence_delay(1000);
21     // execute Texist and Tnew
22     ((void (*)())Texist)(); mfence_delay(1000);
23     ((void (*)())Tnew)(); mfence_delay(1000);
24     // probe the valid state
25     ((void (*)())probe)(); mfence_delay(1000);
26     if (cache_hit(probe_addr)) { count += 1; }
27   }
28   return 1.0f * count / TOTAL;
29 }
30 Texist_state = get_hit_rate(Pexist, pred_exist);
31 Tnew_state = get_hit_rate(Pnew, pred_new);
```

Listing 1. Code that observes the valid states of the existing entry (T_{exist}) and the new entry (T_{new}). The offsets of the two branches are `0xc` and `0xf`.

existing branch as T_{exist} and the newly executed branch as T_{new} in the following sections. In this section, we test whether the two branches remain valid after the BTB updates, using the C code in Listing 1.

In Listing 1, Line 1 and 3 generate two direct jump instructions, `Texist` and `Tnew`, at aligned base addresses with different offsets (`0xc` and `0xf`) and jump lengths (`0x600` and `0xc00`). They represent T_{exist} and T_{new}, respectively. The BTB uses the address of the last byte within branch instructions for indexing (see Section IV-A). Therefore, the addresses of `Texist` and `Tnew` should be reduced by four relative to their offsets, as the direct jump instruction is 5 bytes in length. Moreover, we create two return instructions (`Pexist` and `Pnew`) in Line 5-6, aligned with `Texist` and `Tnew`, to probe the validity of their BTB entries via `get_hit_rate` (Line 30-31). Their predicted targets are `pred_exist` and `pred_new`.

For each test in `get_hit_rate` (Line 18-26), the BTB is flushed using an eviction set (Line 18). Then, in Line 22, `Texist` is executed first to establish the initial BTB state, ensuring an entry already exists in the BTB set. In Line 23, we invoke `Tnew` to introduce a new entry into the BTB, triggering an update mechanism between the two entries. Finally, we call `probe` in Line 25, and observe whether the predicted target (`probe_addr`) is fetched into I-Cache to check the valid state of the tested BTB entry.

We insert `mfence_delay()` to delay subsequent operations between these code segments, ensuring that the executed branch is added to the BTB. Additionally, this ensures a consistent branch history for each indirect call, avoiding mispredictions during each loop execution and thus preventing interference with the experimental results [39].

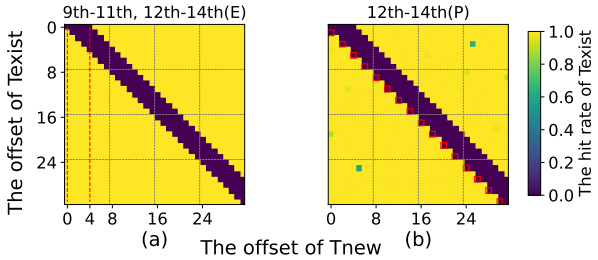

Fig. 6. The hit rate of T_{exist} after sequentially executing T_{exist} and T_{new} with varying offset pairs. In (b), cells with red borders indicate offset pairs with the pseudo-overlapping collision.

We enumerate the offsets of T_{exist} and T_{new} within the range of 0 to 31 and measure the hit rates of the corresponding two entries, demonstrating their validity in the BTB. Figure 6 and 7 illustrate the results.

Validity of T_{exist}. On all processors, the execution of T_{new} can cause T_{exist} to become invalid in the BTB when placed at certain positions (refer to Figure 6). For each offset value of T_{exist}, there are always five consecutive offset values of T_{new} that can invalidate T_{exist} (i.e. Figure 6(a)). In particular, if an address of any byte within T_{new} instruction collides with the existing entry (*i.e.*, have the same index, tag, and offset), the existing entry is invalidated. Therefore, we infer that the existing entry will be invalidated if the newly executed branch instruction overlaps it in the BTB. To further validate this, we repeat the experiment using 2-byte `jmp` instructions for T_{new} and obtain consistent results. We define this as *overlapping collision*.

Additionally, on the performance cores of Intel 12th–14th generation Core processors, we observe an additional behavior (refer to Figure 6(b)): when the offset of T_{exist}, denoted as k, is odd, it can be invalidated by T_{new} with an offset of $k-1$, even though this offset pair does not indicate the overlapping collision. We refer to this as *pseudo-overlapping collision*.

Validity of T_{new}. For T_{new} (refer to Figure 7), almost all offset pairs indicate that T_{new} is valid after its execution. On performance cores, however, T_{new} will not be valid when meeting the pseudo-overlapping collision (i.e. Figure 7(b)). And on efficient cores, T_{new} is invalid when it has the same offset value with T_{exist} (i.e. Figure 7(c)).

Through our results, we conclude that not only can non-branch instructions deallocate branches in the BTB [52], branches may also deallocate a BTB entry when their offsets have the (pseudo) overlapping collision. Meanwhile, newly executed branches may not be inserted in the BTB, which relies on the BTB state.

Observation 2. Newly executed branch instructions can deallocate existing entries at certain offsets and may not be valid in the BTB.

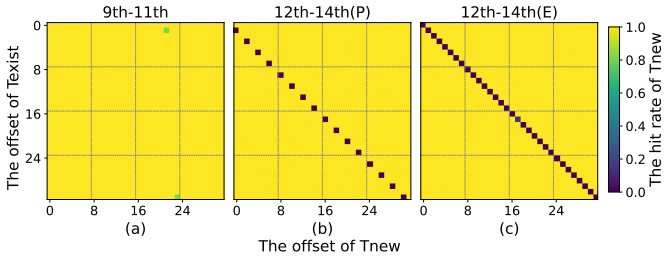

Fig. 7. The hit rate of T_{new} after sequentially executing T_{exist} and T_{new} with varying offset pairs.

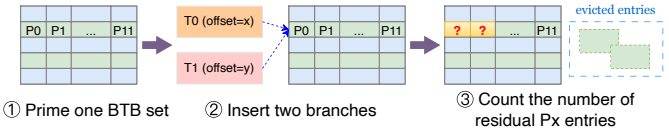

Fig. 8. The experimental design for determining the number of entries evicted by T_{exist} and T_{new}. `P0-P11` represent entries created by `prime` branches. `T0` and `T1` represent two entries created by T_{exist} and T_{new}.

C. Affected Entries during BTB Update

After observing the validity of T_{exist} and T_{new} following BTB updates, we are interested in understanding what actually happens within a BTB set during the update process. For example, when the T_{exist} entry is deallocated, is it directly replaced by the new entry, or is it simply invalidated, after which the new entry is inserted into the BTB by evicting another existing entry?

The key distinction between these two mechanisms is that the latter will affect another existing BTB entry, aside from the deallocated entry. Building on this, we aim to determine how many entries are affected when T_{exist} and T_{new} are sequentially executed, in order to discriminate different types of update logic. We design an experiment, illustrated in Figure 8, that leverages BTB PRIME+PROBE to measure how many entries are evicted by the two branches.

We first fill a BTB set with multiple `prime` branches, each mapped to the same set with a different tag, and equal in number to the BTB's associativity. Subsequently, we invoke T_{exist} and T_{new} with matching indexes and tags to evict a specific number of `prime` branches. Afterward, we probe the BTB set to determine the number of residual `prime` branches still present in the BTB. This count reveals how many entries were evicted by T_{exist} and T_{new}.

The key insight of this experiment is that if T_{new} directly replaces T_{exist}, it will result in the eviction of only one `prime` branch (*i.e.*, T_{exist} evicts one `prime` branch, and T_{new} replaces T_{exist}). Conversely, if T_{exist} is only invalidated but not evicted from the BTB, T_{new} will occupy another entry slot of an existing entry.

The experimental results are summarized in Figure 9, which outlines three cases based on the tested processors:

- **On Core 9th-11th processors:** When T_{exist} and T_{new} share the same offset, only one `prime` branch is evicted from the BTB. Conversely, if their offsets differ, two entry slots are occupied, resulting in the eviction of two `prime` branches.
- **On Core 12th-14th processors, Raptor (Golden) Cove:** The result on Raptor Cove differs slightly from that observed on Core 9th–11th generation processors. Apart from the offsets being equal, if they have the pseudo-overlapping collision, only one `prime` branch gets evicted.
- **On Core 12th-14th processors, Gracemont:** On Gracemont, when the offsets of T_{exist} and T_{new} have the overlapping collision, only one `prime` branch is evicted; otherwise, two are evicted.

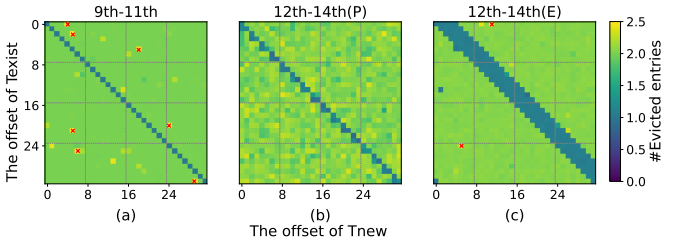

Fig. 9. The average number of evicted `prime` branches under all offset pairs of T_{exist} and T_{new}. Cells marked with red crosses indicate noise points where the value exceeds 2.5.

TABLE II
ALL OFFSET-RELATED BTB UPDATE MECHANISMS. COLUMN T_{exist} AND T_{new} SHOW THE VALID STATES IN EACH MECHANISM. #AFFECTED MEANS THAT THE NUMBER OF AFFECTED ENTRIES AFTER EXECUTING T_{exist} AND T_{new}.

Update Mechanism	T_{exist}	T_{new}	#Affected
Direct Replacement (DR)	✗	✓	1
Invalidation + Allocation (IA)	✗	✓	2
Invalidation (I)	✗	✗	1
Allocation (A)	✓	✓	2

After obtaining the number of evicted entries for all offset pairs of T_{exist} and T_{new}, combining with their respective valid states (Figure 6 and 7), we summarize that there are four cases in the BTB update mechanism (see Table II):

1) ***Direct Replacement*** **(DR)**: the existing entry is directly replaced with the newly executed branch.
2) ***Invalidation + Allocation*** **(IA)**: the existing entry is invalidated, and another BTB entry is allocated for the newly executed branch.
3) ***Invalidation*** **(I)**: the existing entry is invalidated.
4) ***Allocation*** **(A)**: the existing entry is unaffected, with the allocation of a BTB entry for the new branch.

Table III shows the underlying BTB update mechanisms for different offset patterns.

Observation 3. There are 4 different update mechanisms, depending on the offsets of the two branches.

TABLE III
OFFSET-RELATED BTB UPDATE MECHANISMS FOR ALL OFFSET COLLISIONS ACROSS INTEL CORE GENERATIONS, SHOWING CASES WHERE T_{exist} AND T_{new} ARE FROM THE SAME OR DIFFERENT USER PROCESSES ON THE SAME LOGICAL CORE.

Offset collision	9-11	12-14(P)	12-14(E)
overlapping (same offset)	DR	DR	I
overlapping (different offset)	IA	IA	DR
pseudo-overlapping	A	I	A
none	A	A	A

TABLE IV
THE OFFSET-RELATED BTB UPDATE MECHANISMS IN THE K-U AND U-K SCENARIO, EXCEPT ON PERFORMANCE CORES.

Offset collision	9-10 K-U	9-10 U-K	11	12-14(E)
overlapping (same offset)	DR	DR	DR	
overlapping (different offset)	IA	A	A	A
others	A	A	A	

TABLE V
THE OFFSET-RELATED BTB UPDATE MECHANISMS IN THE K-U AND U-K SCENARIO ON PERFORMANCE CORES.

Offset relationship	12-14(P)
same offset	DR
same offset[4:1], different offset[0]	I
other	A

D. Cross Different Domains

Through experiments introduced in Section IV-B and IV-C, we obtain offset-related BTB update mechanisms in a single user process. In this section, we investigate the update mechanism of branches belonging to different execution domains. Specifically, we create T_{exist} and T_{new} in different execution entities and repeat the two experiments in Section IV-B and IV-C to observe the underlying BTB update logic. In addition, we enable all hardware mitigations against the Spectre-v2 attack deployed on all tested processors via Linux default boot parameters. The following sections introduce five different scenarios and their experiment results.

1) Same Logical Core (SLC): In this scenario, T_{exist} and T_{new} belong to two user processes running on the same logical core. The results indicate that the BTB update behavior remains consistent with the findings in Table III, which presents results from experiments conducted within a single user process. This result can be strong evidence that the BTB does not contain the Process Identifier (PID) in entries since no special update logic exclusively serves for entries belonging to different processes [28].

2) Simultaneous Multithreading (SMT): When T_{exist} and T_{new} are executed by two sibling threads on the same physical core, all offset pairs consistently exhibit the *Allocation* update mechanism, with exactly two entries being evicted. This behavior suggests that the BTB is logically isolated but physically shared between SMT threads, as mutual evictions can still occur.

3) Cross Physical Core (CPC): When T_{exist} and T_{new} are executed by threads on different physical cores, the *Allocation* mechanism still applies, but only a single entry is evicted. This behavior supports the conclusion that the BTB is physically isolated across physical cores, with each core maintaining its own private BTB.

4) Kernel v.s. User (K-U & U-K): These scenarios examine branch interactions across user and kernel modes. In the K-U scenario, T_{exist} is a kernel-space branch and T_{new} is a user-space branch, whereas in the U-K scenario, T_{exist} is in user space and T_{new} is from kernel space. All `prime` branches are user-space branches in both scenarios. The corresponding results are presented in Table IV and Table V.

Across all evaluated microarchitectures except Gracemont, *Direct Replacement* mechanism is applied when T_{exist} and T_{new} have the same offset, although user branches cannot use entries created by kernel branches on some architectures, according to our experiments in Section III-A. In addition, on 9th-10th Intel Core processors, when offsets have the overlapping conflict, user branches can deallocate kernel branches, applying *Invalidation* + *Allocation* mechanism, but not vice versa. And on 11th-generation processors, different offsets point to the *Allocation* mechanism. On Gracemont, all offset pairs point to the *Allocation* mechanism.

On performance cores (as shown in Table V), when T_{exist} and T_{new} share the same `offset[4:1]` but differ in `offset[0]`, the BTB applies the *Invalidation* update mechanism. In all other cases, the *Allocation* mechanism is used.

E. Root Cause of JumpOverASLR

Building on our findings, we analyze the root cause of JumpOverASLR [15]. JumpOverASLR exploits branch collisions to infer victim branch addresses and successfully breaks (K)ASLR on Intel Haswell processors. However, this work did not uncover that the BTB uses the address of the last byte within the instruction for indexing, and instead aligned branch instruction addresses to induce collisions. This means that the underlying BTB update mechanism in this attack implicitly depends on the instruction length, which determines the last byte address.

We replicate the experiments described in Section IV-B on an Intel Core i7-4790 processor (Haswell microarchitecture). The result shows that when two user branches (or one user branch and another kernel branch), belonging to the same logical core, meet the overlapping collision, they can deallocate each other, and the user branch can use BTB entries of another user or kernel branch for prediction. Therefore, in the attack procedure of JumpOverASLR, if the instruction length of the attacker's branch exceeds that of the victim's branch, the BTB applies the *Allocation* mechanism after the execution of the victim's branch, and the attacker will use the entry of the victim's branch to generate the predicted target due to the lower offset of the victim's entry [52]. In this case, the attacker's entry remains *valid* but is not hit by the attacker.

Conversely, if the victim's branch is longer or has the same length as the attacker's branch, it can deallocate the entry of the attacker's branch, and only the victim's entry can be used for prediction. In both cases, the attacker hits the victim's entry for branch prediction, which causes the timing slowdown. Based on these findings, we categorize JumpOverASLR as an access-based attack.

Furthermore, JumpOverASLR may not succeed in the cross-privilege scenario on Intel Core processors starting from the 11th generation, where user and kernel branches of different lengths result in mismatched offsets, thereby preventing BTB collisions. In contrast, our work thoroughly investigates the offset-related BTB update mechanism and introduces a deterministic BTB side-channel, as detailed in Section V.

V. Evaluation of Occupy+Probe

Based on previous reverse-engineering results, in this section, we detail Occupy+Probe, a cross-privilege BTB side-channel attack that leaks kernel branch information from user mode, and evaluate it experimentally in the following sections.

A. Experiment Setup

We perform our experiments on processors running the Ubuntu 20.04 or 22.04 operating system and enable hardware mitigations against the Spectre-v2 attack. We simulate the kernel context via a Linux kernel module for each experiment. We set a conditional branch in the kernel module, and its condition relies on a secret value. For experiments described in Section V-B and V-C, the objectives are to leak the condition through Occupy+Probe. In Section V-D, we utilize Occupy+Probe to leak the address information of kernel branches to break KASLR.

B. Breaking Hardware Isolation

As described in Section III-C, our approach requires crafting an *occupy* branch whose corresponding BTB entry will be directly replaced if the target kernel branch is executed. Based on our findings in Section IV-D, BTB entries with identical index, tag, and offset can replace one another, even when belonging from different privilege levels. To target a specific kernel branch, we construct a user-space *occupy* branch whose last byte address shares the lowest 36 bits with that of the kernel branch, as the BTB uses this byte for indexing, thereby enabling the *Direct Replacement* update mechanism to be triggered. By following the workflow illustrated in Figure 4, we can determine whether the monitored kernel branch has been executed or taken.

To evaluate the attack performance of Occupy+Probe, we implement the workflow shown in Figure 4, with the victim code residing in a Linux kernel module. We randomly generate 1,000 secret bits, each of which is used as the condition for a conditional jump in the kernel. The kernel branch is triggered via an `ioctl` syscall, after which we probe the validity of the *occupy* branch to infer the corresponding secret bit. Figure 10 presents a portion of the results on Intel Core i9-14900K, showing the access times to the predicted target (*i.e.*, `Predicted Target` in the probe phase) across iterations. Low access times indicate that the *occupy* branch remains in the BTB due to the prefetching of the predicted target, implying the kernel branch was not taken (*i.e.*, the secret bit is one). Conversely, high access times suggest that the kernel branch was taken, evicting the *occupy* entry (*i.e.*, the secret bit is zero). As demonstrated, Occupy+Probe successfully reveals the execution outcome of the kernel branch, even on processors equipped with cross-privilege hardware isolation. The overall attack accuracy on various processors is reported in the "Occupy+Probe, w/o CDB" column of Table VI, as the experiment described in this section is part of the evaluation in Section V-C.

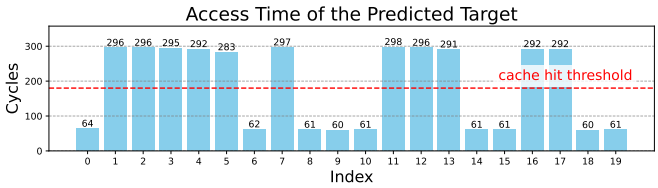

Fig. 10. The Occupy+Probe result, showing the access time of `Predicted Target` after executing `Probe`. The secret bits are 10000010111000110011.

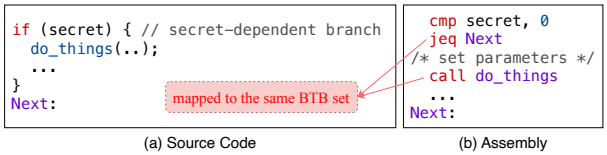

Fig. 11. The example code that prevents BTB Prime+Probe attacks.

C. High Spatial Resolution

Now we evaluate the spatial resolution of Occupy+Probe. As described in Section III-A, existing eviction-based BTB attacks (*i.e.*, BTB Prime+Probe attack) [1]–[3], [54] cannot leak finer execution results in a 32-byte-aligned code block, for example, which branch instruction is executed. This shortcoming may cause the BTB Prime+Probe attack to fail to leak the secret value. Take the code in Figure 11 as an example. If the secret value is not zero, the function `do_things` will be invoked. From the compiled assembly code, we can discover that, if the condition jump instruction and call instruction map to the same BTB set, there will be always one branch inserted in the BTB set, regardless of the secret value. Therefore, BTB Prime+Probe attack can only observe that one `prime` branch is evicted, but cannot obtain which victim branch is executed, thereby preventing inference of the secret value (see Figure 12). Additionally, inserting a confusing dummy branch after the conditional branch, similar to the code in Figure 11, can be a proper mitigation intentionally deployed in security-sensitive code to defend against BTB Prime+Probe attacks. If so, the BTB Prime+Probe attack would no longer be effective.

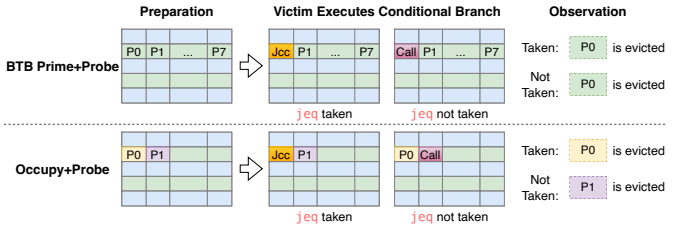

Fig. 12. Comparison between the BTB PRIME+PROBE and OCCUPY+PROBE attack.

TABLE VI
SUCCESS RATE WITH/WITHOUT THE CONFUSING DUMMY BRANCH (CDB).

Processor	BTB PRIME+PROBE		OCCUPY+PROBE	
	w/o CDB	*w/ CDB*	*w/o CDB*	*w/ CDB*
i7-9700	100%	48.9%	100%	100%
i9-10850K	100%	48.9%	100%	100%
i7-11700K	85.5%	49.1%	96.4%	91.9%
i7-12700K	100%	48.9%	99.4%	100%
i9-13900K	99.8%	48.9%	99.1%	98.6%
i9-14900K	100%	48.9%	100%	100%

In contrast, our OCCUPY+PROBE attack effectively addresses this problem. Due to the special *Direct Replacement* BTB update mechanism, each branch can directly evict the BTB entry with identical index, tag, and offset, regardless of the replacement policy. Therefore, for the code in Figure 11, OCCUPY+PROBE attack can generate an *occupy* branch for each victim branch mapped in one BTB set, and trace their execution results individually, as shown in Figure 12. After the victim execution, the attacker probes each *occupy* branch to obtain its validity, thus inferring the execution result of the corresponding victim branch. Therefore, through leaking finer execution results within one BTB set, the attacker successfully reveals secret values in Figure 11.

Evaluation. We evaluate the attack performance between the BTB PRIME+PROBE attack and OCCUPY+PROBE attack, with the victim code containing a secret-dependent conditional branch and a confusing dummy branch mapped to the same BTB set (similar to Figure 11). We randomly generate 1,000 secret bits and then count the number of bits each attack successfully leaks. Table VI shows the success rate of the two attacks with/without the confusing dummy branch on various processors. Without the confusing dummy branch, the BTB PRIME+PROBE attack can successfully leak secret values, as demonstrated in previous works. However, when inserting the confusing dummy branch, the BTB PRIME+PROBE attack does not work, while OCCUPY+PROBE can still leak secret values.

D. Leaking the Tag

In this section, we demonstrate the capability of OCCUPY+PROBE to leak tag values of victim branches, relying on the condition that replacement occurs only when both the tag and offset match between the attacker and victim branches. To illustrate this capability, we exploit OCCUPY+PROBE to break KASLR on the i7-11700K processor.

KASLR. Kernel Address Space Layout Randomization, *i.e.*, KASLR, was introduced in Linux version 3.14 to randomize kernel addresses. KASLR aims to mitigate code-reuse attacks and other similar threats, which rely on specific kernel runtime addresses, by randomly generating kernel addresses at boot time. In the Linux kernel implementation, address bits [29:21] are randomized, with all kernel addresses maintaining the same offset from the base. As a result, leaking a single kernel address is sufficient to break KASLR. Our attack aims to leak the virtual address of a specific kernel branch.

Attack Description. Our experiment runs on Ubuntu 22.04.5 LTS with the kernel version 6.8.0-60-generic. We choose a `call` branch in the kernel image of the `getpid` syscall as the monitored branch. We guess the tag of this branch and create an aliased branch in user mode, and use OCCUPY+PROBE to verify the guess. If the guess is correct, the allocated kernel entry will directly replace the user branch. Note that on i7-11700K, the tag field records address bits [33:14] with folded xor operation, and KASLR only randomizes address bits [29:21]. It can be deduced that the correct guess of address bits [29:21] equals the match of the tag field between the guessed branch and the kernel branch, as the tag field contains enough entropy of random bits.

Evaluation. We reboot the system 10 times, each time generating a different random address for the kernel image. For each reboot, we execute our attack 100 times to attempt to leak the kernel address, resulting in a total of 1,000 trials. Our attack achieves a success rate of 97.5%. We further evaluate our approach on other processors listed in Table I, where we successfully leak the tag value associated with a kernel branch. While the leaked tag does not directly reveal the full kernel address, it narrows down the set of possible addresses. Specifically, the leaked tag narrows the kernel address space from 2^9 to 2^6 candidates on Core i9-14900K, reducing the search space by a factor of 8.

VI. ATTACKING LINUX KERNEL CRYPTO API

To showcase the practical applicability of OCCUPY+PROBE, we present a case study in which the attack is deployed in a real-world scenario. Specifically, we target the Linux Kernel Crypto API [31], a widely used library that provides a comprehensive set of cryptographic ciphers within the kernel. This attack aims to leak the private key used in the RSA algorithm.

A. RSA

RSA [46] is a typical asymmetric cryptosystem, which exploits the public key for encryption and signature verification, and the private key for decryption and signing. The public key consists of e and n, where e is the exponent and n is the multiplication of two big primes p and q, and the private key is a mathematically derived number d satisfying $e \times d \equiv 1 (\text{mod } \phi(n))$. In the procedure of encryption, the ciphertext c is calculated through $c = m^e \text{ mod } n$, where

m is the message to be encrypted. In the decryption, the message can be resolved through $m = c^d \bmod n$. In implementations of the RSA algorithm, most cryptographic libraries apply an optimization to the decryption procedure, with precalculating d_p, d_q, and q_{inv}, where $d_p = d \bmod (p-1)$, $d_q = d \bmod (q-1)$, and $q_{inv} = q^{-1} \bmod p$. The algorithm first calculates $m_p = c^{d_p} \bmod p$ and $m_q = c^{d_q} \bmod q$, and then uses the Chinese Remainder Theorem (CRT) [43] to get the final result m.

In addition, the square-and-multiply algorithm [44] is often exploited to speed up the modular exponentiation like $a^b \bmod n$. The core idea is converting the exponent to its binary representation and processing it bitwise. For each bit, the current result is squared, and if the bit is 1, the result is also multiplied by the base. Since this approach significantly reduces the computational complexity from linear to logarithmic with respect to the exponent value, it exposes an attack surface to control-flow side-channels [1]–[3], [12], [16], [27], [29], [33], [50], [52], [54], as the exponentiation introduces a secret-dependent branch, whose execution results can be leaked to recover the secret key.

B. Linux Kernel Crypto API

The Linux Kernel Crypto API [31] was introduced in kernel version 2.5.45, offering a range of cryptographic algorithms within the kernel. Its primary goal is to provide fast, general-purpose cryptographic operations for the kernel and its components. This API is utilized by various applications, including cryptographic filesystems and IPSec. It supports a wide array of cryptographic algorithms, such as block ciphers, hash functions, and asymmetric ciphers, including RSA. The implementation of the RSA algorithm in the Linux kernel applies the optimization of both the CRT and the square-and-multiply algorithm.

The `mpi_powm` function implements the square-and-multiply algorithm, as illustrated in Listing 2. These two loops traverse all bits in `exp`, and apply multiplications upon meeting one (Line 5). As describe in Section VI-A, the parameter `exp` is passed with d_p and d_q respectively, which is contained in the private key. Therefore, the branch instruction corresponding to the `if` statement is secret-dependent, and we seek to leak the execution result of this branch in each loop, hence recovering d_p and d_q.

```
1  // RES = BASE ^ EXP mod MOD
2  int mpi_powm(MPI res, MPI base, MPI exp, MPI mod) {
3    for (;;) {
4      while (c) {
5        if ((mpi_limb_signed_t) e < 0) {
6          /* mpihelp_mul( xp, rp, rsize, bp, bsize ); */
7          ...
8        }
9        e <<= 1; c--;
10       cond_resched();
11     }
12     i--;
13     if (i < 0) break;
14     e = ep[i]; c = BITS_PER_MPI_LIMB;
15 }}
```

Listing 2. A simplified square-and-multiply implementation in `mpi_powm` (*lib/crypto/mpi/mpi-pow.c* in Linux v6.8 kernel source code). `MPI` is an internal data type of big integers.

C. Attack Procedure and Challenges

Attack Scenario. We instantiate a Linux kernel module as the victim, embedding an asymmetric key pair internally and exposing an interface for signing messages from user-space applications using the private key through the Linux Kernel Crypto API. The private key is inaccessible to user space, ensuring that cryptographic operations can only be performed within the restricted kernel context, and user-space threads issue a signing request through an `ioctl` system call. This model reflects specific security-sensitive systems, which support secure mechanisms for device authentication, capability licensing, and data integrity assurance by confining sensitive cryptographic operations to the kernel space [22], [25], [30].

As the attacker thread runs on the same logical core as the decryption thread, to leak the secret bits across all iterations of the `mpi_powm` function (Lines 4–11 in Listing 2), the attacker operates in an interleaved mode with the victim. This requires frequently preempting the decryption process to monitor the execution of the secret-dependent branch in each iteration.

Attack Procedure. We design the attack procedure illustrated in Figure 13. It involves two user-space threads: the *attacker*, which performs OCCUPY+PROBE attack, and the *requester*, which issues signing requests to the victim kernel module. The kernel context includes both the implementation of the Linux Kernel Crypto API and the instantiated victim kernel module.

First, the attacker generates an *occupy* branch monitoring the target kernel branch, executes it to insert a BTB entry (❶), and subsequently yields the CPU, allowing the requester thread to proceed. The requester issues an `ioctl` syscall, which triggers the invocation of the RSA decryption function (❷). This function internally calls `mpi_powm` twice, using the secret exponents `exp` = d_p and `exp` = d_q respectively. Once this function completes one iteration (❸), the attacker interrupts the execution of `mpi_powm` (❹) and probes the *occupy* branch to determine whether the secret-dependent branch was taken (❺), thus inferring the secret bit processed in the current iteration. After probing, the attacker re-executes the *occupy* branch to restore the BTB entry, yields the CPU (❻) to let `mpi_powm` proceed to the next iteration, transitioning to Step ❸. We regard the completion of steps ❸ - ❻ as a single *pass*. After all iterations are complete, the attacker reconstructs the private key components d_p and d_q through leaked execution traces of all passes.

Challenges. There are two challenges in the attack procedure:

1) How can the attacker periodically interrupt the RSA decryption operation in user mode (*i.e.*, without kernel privilege)?
2) How can it be determined whether an iteration in `mpi_powm` has been completed in a single pass, thereby avoiding spurious results potentially caused by overly

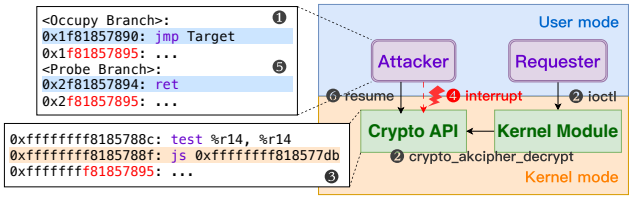

Fig. 13. The attack procedure.

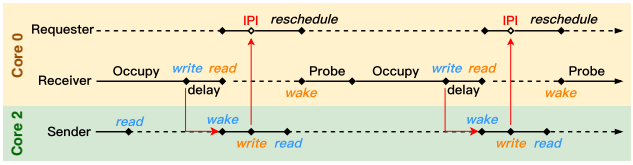

Fig. 14. The timeline of the attack. The `read` and `write` with blue color indicate operations to *synefd*, and those with orange color indicate *intefd*.

frequent interrupts or other reasons?

Section VI-D and VI-E describe our proposed methods to overcome these two challenges.

D. Periodic Interrupts

In this section, we introduce our method that achieves periodic interrupts to the loop execution in Listing 2 in user mode. Prior works [6], [19], [35], [40], [56] successfully interrupt the victim execution of user programs with a high resolution. However, these methods are not suitable in our scenario since the victim is in kernel mode. Most mainstream Linux distributions configure the kernel with setting `CONFIG_PREEMPT_VOLUNTARY` by default [8], [17], [37]. Under this configuration, the kernel only yields the CPU at explicitly defined preemption points, making it infeasible to forcefully interrupt the kernel execution from user space in a fine-grained and timely manner as required by prior preemption-based methods.

`cond_resched`. The loop within `mpi_powm` invokes the function `cond_resched` at the end of each iteration (Line 10). This function checks whether the current task should be rescheduled by examining the per-CPU variable `preempt_count`. If this value is zero, the function triggers a context switch, allowing the CPU to schedule another runnable task. This mechanism ensures that, on non-preemptible kernels, long-running computations in `mpi_powm` do not block task scheduling and degrade system responsiveness [10].

This function provides an opportunity for the attacker to interrupt kernel execution at controlled points. By ensuring that `preempt_count` remains zero during each loop iteration, the attacker can periodically preempt the decryption routine without requiring kernel privileges. However, without deliberate intervention by the attacker, the decryption routine is rarely interrupted, except occasionally by timer interrupts, as shown in our experiments. Therefore, the attacker must actively induce context switches to achieve frequent preemption.

Interrupting via eventfd. To orchestrate the interrupt mechanism, we create two attack threads, referred to as *sender* and *receiver*. The receiver thread is bound to the same logical core as the requester thread, while the sender thread runs on a separate physical core. Both threads share an event file descriptor (`eventfd`) [23], which serves as a primitive for event-based wait and notification.

Initially, the sender does not write to the eventfd. When the receiver attempts to read from it, the read operation blocks, putting the receiver thread to sleep and causing the operating system to schedule the requester thread to run on the shared core.

At a designated time, the sender writes to the eventfd, which wakes up the receiver, and the sender core will send a reschedule inter-processor interrupt (IPI) to the receiver core. This interrupt temporarily halts the execution of the requester and resets its `preempt_count` to zero in the interrupt handler, and then resumes the requester execution. Subsequently, when the requester reaches the `cond_resched` call, the rescheduling mechanism is triggered, and the receiver thread is scheduled. Through this coordination, the execution of `mpi_powm` can be interrupted once per iteration.

We deploy OCCUPY+PROBE within the receiver thread. Before reading from the eventfd, the receiver executes the *occupy* branch to insert a BTB entry. Once it is scheduled and regains control, the receiver probes the *occupy* branch to infer the victim branch execution result.

Synchronization. One key point of the above process is to precisely write to the eventfd in the sender thread. It must be written exactly between the receiver's read operation and the requester's completion of the current iteration. Therefore, it is necessary to synchronize the sender and receiver threads. We introduce another eventfd, referred to as *synefd*, while the aforementioned one for interrupting the requester is referred to as *intefd*. Initially, the sender blocks by reading from *synefd*, waiting for the receiver to complete the occupy phase. Once completed, the receiver writes to *synefd* to wake up the sender, and then reads from *intefd*, causing it to sleep and allowing the requester to begin execution. After the sender wakes up, it writes to *intefd* to wake up the receiver, interrupting the execution of the requester and setting the `preempt_count` to zero. The entire timeline of the three threads is shown in Figure 14.

Additionally, we observe that if the receiver reads from *intefd* immediately after writing to *synefd*, the requester may execute for an excessively long duration, resulting in multiple iterations of `mpi_powm` being executed within a single pass. To prevent this, the receiver introduces a short delay between writing to *synefd* and reading from *intefd* in the receiver thread. We also utilize the `nice` command in Linux, which is available in user mode, to lower the priority of the requester process. This adjustment alleviates the starvation issue by making the requester more likely to be preempted when the attacker triggers an interrupt.

E. Discarding Redundant Bits

Experimental results reveal that, in some cases, the receiver reads from *intefd* after the sender has already written to it. This timing mismatch causes the eventfd read operation to return immediately without blocking, preventing the requester from being scheduled during that pass. As a result, redundant bits may be included in the results. To address this issue, we monitor another branch within the `mpi_powm` loop guaranteed to execute in every iteration. Specifically, we leverage the `cond_resched` function call, which corresponds to a `call` instruction and is executed unconditionally, regardless of the secret bit. The attacker introduces an additional *occupy* branch to track this `call` instruction, enabling the detection of whether an iteration has actually been executed. This allows the attacker to discard bits leaked during passes in which no iteration occurs, thereby improving the accuracy of the extracted data.

F. Results

We conduct our attack on the Intel i9-14900K processor running Ubuntu 22.04.5 LTS with the Linux 6.8.0-51-generic kernel, configured with default settings. All hardware-based defenses against Spectre-v2 are enabled. In the kernel image, the secret-dependent branch within the `mpi_powm` is compiled into two distinct conditional branches, corresponding to two separate execution paths. Therefore, both branches must be tracked to infer the secret bits based on their respective execution results. Prior to the experiment, we obtain the addresses of the two secret-dependent conditional branches and the `call` instruction mentioned in Section VI-E.[3]

In our experiment, we first launch the attack process, namely the sender and receiver threads. These two threads initiate the interrupt mechanism via `eventfd`. Subsequently, we start the requester thread, which triggers the RSA decryption. During execution, the decryption routine is frequently interleaved by the receiver thread. The sender and receiver threads totally perform 2×10^6 passes, during which the requester thread is executed 10 times. After the attacker processes are complete, we collect all leaked information, including the execution outcomes of the secret-dependent branches and the `call` instruction.

We count the number of iterations detected within the first 5,000 passes for each pass, and the result is illustrated in Figure 15. There are 9 spikes in the figure, each corresponding to one execution of the requester thread. This suggests that one execution may not have been successfully interrupted. Among these spikes, the highest few values slightly exceed 2,000, which is close to the total number of executed iterations, indicating that nearly all iterations are successfully interrupted.

For each spike, we extract the leaked secret bits and obtain 9 sequences of bits. We employ the Levenshtein distance [45] to compare the leaked bit sequence against the ground-truth

[3]In our threat model, we assume the attacker has knowledge of the target branch address. Even if this assumption does not hold, the attacker can still infer the tag of the target branches using the method described in Section V-D, which is sufficient for the attack to succeed.

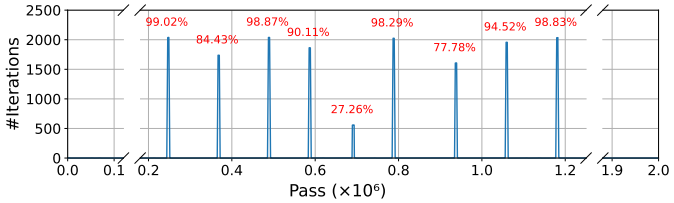

Fig. 15. Number of detected iterations within the first 5000 passes. The percentage value above each spike represents the accuracy of the corresponding sequence.

secret bit sequence obtained from actual branch execution (*i.e.*, d_p and d_q). The accuracy of the leaked sequence is quantified using Formula (1) based on this distance metric. Figure 15 demonstrates that all 9 sequences are capable of leaking partial secret bits, with the highest few achieving an accuracy exceeding 98%. We further execute the requester thread 150 times to measure the mean and standard error of the attack accuracy. The results show a mean accuracy of 98.6% with a standard error of 4.7%, demonstrating the effectiveness of this attack.

$$\text{Accuracy} = \left(1 - \frac{\text{Levenshtein}(S_{\text{leak}}, S_{\text{true}})}{\max(|S_{\text{leak}}|, |S_{\text{true}}|)}\right) \times 100\% \quad (1)$$

VII. Discussion

A. Mitigations

Hardware defenses. A key insight of the Occupy+Probe attack is that the *Direct Replacement* BTB update mechanism remains across different privilege domains. In particular, any mechanism through which kernel branches can affect user-space BTB entries in an observable way (i.e., by invalidating or directly replacing them such that their effects can later be detected from user space) can serve as the attack primitive of Occupy+Probe. Therefore, to mitigate Occupy+Probe, one potential defense is to disable these mechanisms entirely, thereby strengthening cross-privilege isolation. Both prior work [7], [28], [54] and our experimental findings indicate that BTB entries record the privilege level at which branch instructions are executed, which suggests a viable solution against Occupy+Probe.

Additionally, flushing the BTB on context switches offers an alternative defense against Occupy+Probe and other BTB-based cross-privilege attacks. However, this approach incurs substantial performance overhead, as it eliminates executed branch information, disabling branch prediction for both user and kernel code [27].

Software defenses. Data-oblivious programming [9], [27], [34], [51] can be an effective approach to eliminate secret-dependent control-flow information in secret-sensitive codes. If so, all control-flow leakage attacks exploiting branches cannot successfully leak the secret in the victim program. However, achieving data-oblivious programming requires sophisticated efforts to modify the application code, and for

complicated applications, eliminating branches may incur significant overhead.

On processors with hybrid architectures, such as the 12th to 14th generation Intel Core processors, it is advisable to execute sensitive operations on efficient cores, where the *Direct Replacement* mechanism is absent and OCCUPY+PROBE is therefore ineffective. However, this defense may result in lower performance than running on performance cores [13].

For Linux Kernel Crypto API, the `cond_resched` can be removed to prevent frequent interrupts to the decryption operation but may cause starving issues, violating the original purpose of adding this function [10].

B. Limitations and Future Work

One limitation of OCCUPY+PROBE is that the attack is constrained to the same logical core and cannot be extended across SMT threads or physical cores, as the *Direct Replacement* mechanism is not present in those scenarios. Furthermore, unlike access-based attacks, OCCUPY+PROBE cannot reveal the target address stored in a BTB entry. In addition, our experiments are conducted exclusively on Intel processors. Processors from other vendors, such as AMD [4], [42], [48], ARM [5], and Apple [36], also implement BTBs, which may exhibit similar behaviors to what we have observed on Intel processors. We plan to extend our investigation to these platforms in future work.

VIII. CONCLUSION

In this paper, we present OCCUPY+PROBE, a novel cross-privilege, eviction-based BTB side-channel attack. OCCUPY+PROBE overcomes two key limitations of prior work: access-based BTB attacks are constrained by the privilege boundary due to hardware isolation, while existing eviction-based approaches suffer from limited spatial resolution. To support OCCUPY+PROBE, we reverse-engineer the offset-related BTB update behavior across multiple Intel Core processor generations and uncover the *Direct Replacement* mechanism, which holds even in cross-privilege scenarios. Leveraging this mechanism, OCCUPY+PROBE can infer whether a secret-dependent kernel branch is executed or taken, and can reveal the tag of kernel branches in the BTB, thereby breaking KASLR on Intel Core i7-11700K processors. We further demonstrate an end-to-end attack against the Linux Kernel Crypto API, successfully leaking a private key with over 98% accuracy, according to our experimental evaluation.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their valuable comments and suggestions. This work is partially supported by the National Natural Science Foundation of China (Grant No. 62372258).

REFERENCES

[1] O. Aciiçmez, c. K. Koç, and J.-P. Seifert, "On the power of simple branch prediction analysis," in *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 312–320. [Online]. Available: https://doi.org/10.1145/1229285.1266999

[2] O. Acıiçmez, S. Gueron, and J.-P. Seifert, "New branch prediction vulnerabilities in openssl and necessary software countermeasures," in *Cryptography and Coding*, S. D. Galbraith, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 185–203.

[3] O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert, "Predicting secret keys via branch prediction," in *Topics in Cryptology – CT-RSA 2007*, M. Abe, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 225–242.

[4] Advanced Micro Devices, Inc., *Software Optimization Guide for AMD Family 15h Processors*, 3rd ed., Santa Clara, CA, USA, Jan. 2014, publication No. 47414. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/archived-tech-docs/software-optimization-guides/47414_15h_sw_opt_guide.pdf

[5] Arm Limited, *Arm® Cortex®-X4 Core Technical Reference Manual*, Cambridge, UK, Aug. 2024, document Number: 102484, Version: r0p0. [Online]. Available: https://developer.arm.com/documentation/102484/latest/

[6] C. Ashokkumar, R. P. Giri, and B. Menezes, "Highly efficient algorithms for aes key retrieval in cache access attacks," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2016, pp. 261–275.

[7] E. Barberis, P. Frigo, M. Muench, H. Bos, and C. Giuffrida, "Branch history injection: On the effectiveness of hardware mitigations against Cross-Privilege spectre-v2 attacks," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 971–988. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/barberis

[8] E. Barbieri. (2022) Low latency linux for industrial embedded systems – part ii. [Online]. Available: https://ubuntu.com/blog/industrial-embedded-systems-ii

[9] D. J. Bernstein and B.-Y. Yang, "Fast constant-time gcd computation and modular inversion," *IACR transactions on cryptographic hardware and embedded systems*, pp. 340–398, 2019.

[10] E. Biggers and G. Kroah-Hartman. (2017) lib/mpi: call cond_resched() from mpi_powm() loop. [Online]. Available: https://git.toradex.com/cgit/linux-toradex.git/commit/lib?h=toradex_imx_4.9.123_imx8mm_ga-bring_up&id=443d26a6f791506ca6fdab58b355ba7e170f741e

[11] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T.-H. Lai, " SgxPectre: Stealing Intel Secrets From SGX Enclaves via Speculative Execution ," *IEEE Security & Privacy*, vol. 18, no. 03, pp. 28–37, May 2020. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/MSEC.2019.2963021

[12] Y. Chen, L. Pei, and T. E. Carlson, "Afterimage: Leaking control flow data and tracking load operations via the hardware prefetcher," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 16–32. [Online]. Available: https://doi.org/10.1145/3575693.3575719

[13] I. Corporation. How intel® core™ processors work. [Online]. Available: https://www.intel.com/content/www/us/en/gaming/resources/how-hybrid-design-works.html

[14] ——. (2021) Speculative execution side channel mitigations. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/speculative-execution-side-channel-mitigations.html

[15] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over aslr: Attacking branch predictors to bypass aslr," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13.

[16] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 693–707. [Online]. Available: https://doi.org/10.1145/3173162.3173204

[17] M. Fleming. (2019) Linux kernel preemption and the latency-throughput tradeoff. [Online]. Available: https://www.codeblueprint.co.uk/2019/12/23/linux-preemption-latency-throughput.html

[18] D. Gens, O. Arias, D. Sullivan, C. Liebchen, Y. Jin, and A.-R. Sadeghi, "Lazarus: Practical side-channel resilient kernel-space randomization," in *Research in Attacks, Intrusions, and Defenses*, M. Dacier, M. Bailey, M. Polychronakis, and M. Antonakakis, Eds. Cham: Springer International Publishing, 2017, pp. 238–258.

[19] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games – bringing access-based cache attacks on aes to practice," in *2011 IEEE Symposium on Security and Privacy*, 2011, pp. 490–505.

[20] B. D. Hoyt, G. J. Hinton, D. B. Papworth, A. K. Gupta, M. A. Fetterman, S. Natarajan, S. Shenoy, and R. V. D'sa, "Method and apparatus for implementing a set-associative branch target buffer," Nov. 12 1996, uS Patent 5,574,871.

[21] T. Huo, X. Meng, W. Wang, C. Hao, P. Zhao, J. Zhai, and M. Li, "Bluethunder: A 2-level directional predictor based side-channel attack against sgx," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2020, no. 1, p. 321–347, Nov. 2019. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8401

[22] M. Kerrisk. (2011) ip-xfrm(8) — linux manual page. [Online]. Available: https://man7.org/linux/man-pages/man8/ip-xfrm.8.html

[23] ——. (2024) eventfd(2) — linux manual page. [Online]. Available: https://man7.org/linux/man-pages/man2/eventfd.2.html

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

[25] I. Korchagin, "What is linux kernel keystore and why you should use it in your next application," in *SREcon23 Asia/Pacific*. Singapore: USENIX Association, Jun. 2023.

[26] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *12th USENIX Workshop on Offensive Technologies (WOOT 18)*. Baltimore, MD: USENIX Association, Aug. 2018. [Online]. Available: https://www.usenix.org/conference/woot18/presentation/koruyeh

[27] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside SGX enclaves with branch shadowing," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 557–574. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-sangho

[28] L. Li, H. Yavarzadeh, and D. Tullsen, "Indirector: High-Precision branch target injection attacks exploiting the indirect branch predictor," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 2137–2154. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/li-luyi

[29] C. Liu, S. Feng, Y. Li, D. Wang, W. He, Y. Lyu, and T. E. Carlson, "Mdpeek: Breaking balanced branches in sgx with memory disambiguation unit side channels," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 622–638. [Online]. Available: https://doi.org/10.1145/3676641.3716004

[30] MIT Kerberos Team, "Mit kerberos documentation: Credential cache," https://web.mit.edu/kerberos/krb5-1.12/doc/basic/ccache_def.html, 2013, accessed: 2025-04-14.

[31] S. Mueller and M. Vasut. Linux kernel crypto api. [Online]. Available: https://www.kernel.org/doc/html/v4.17/crypto/index.html

[32] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of aes," in *Topics in Cryptology – CT-RSA 2006*, D. Pointcheval, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–20.

[33] I. Puddu, M. Schneider, M. Haller, and S. Capkun, "Frontal attack: Leaking Control-Flow in SGX via the CPU frontend," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 663–680. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/puddu

[34] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital Side-Channels through obfuscated execution," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 431–446.

[35] B. Roy, R. P. Giri, A. C., and B. Menezes, "Design and implementation of an espionage network for cache-based side channel attacks on aes," in *2015 12th International Joint Conference on e-Business and Telecommunications (ICETE)*, vol. 04, 2015, pp. 441–447.

[36] H. Suzuki, "Brief notes on apple m1 firestorm microarchitecture," https://github.com/ocxtal/insn_bench_aarch64/blob/master/optimization_notes_apple_m1.md, 2022, accessed: 2025-04-13.

[37] L. Torvalds. (2024) Linux kernel x86_64 defconfig. [Online]. Available: https://github.com/torvalds/linux/blob/v6.8/arch/x86/configs/x86_64_defconfig#L7

[38] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient cache attacks on aes, and countermeasures," *Journal of Cryptology*, vol. 23, pp. 37–71, 2010.

[39] D. Trujillo, J. Wikner, and K. Razavi, "Inception: Exposing new attack surfaces with training in transient execution," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 7303–7320. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/trujillo

[40] D. Tsafrir, Y. Etsion, and D. G. Feitelson, "Secretly monopolizing the cpu without superuser privileges." in *USENIX Security Symposium*, 2007, pp. 239–256.

[41] V. Uzelac and A. Milenkovic, "Experiment flows and microbenchmarks for reverse engineering of branch predictor structures," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, 2009, pp. 207–217.

[42] P. Wieczorkiewicz. (2022) The amd branch (mis)predictor: Just set it and forget it! [Online]. Available: https://grsecurity.net/amd_branch_mispredictor_just_set_it_and_forget_it

[43] Wikipedia contributors, "Chinese remainder theorem — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Chinese_remainder_theorem&oldid=1283536923, 2025, [Online; accessed 12-April-2025].

[44] ——, "Exponentiation by squaring — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Exponentiation_by_squaring&oldid=1277105037, 2025, [Online; accessed 12-April-2025].

[45] ——, "Levenshtein distance — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Levenshtein_distance&oldid=1279737312, 2025, [Online; accessed 12-April-2025].

[46] ——, "Rsa cryptosystem — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=RSA_cryptosystem&oldid=1284724493, 2025, [Online; accessed 12-April-2025].

[47] J. Wikner and K. Razavi, "RETBLEED: Arbitrary speculative code execution with return instructions," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3825–3842. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/wikner

[48] J. Wikner, D. Trujillo, and K. Razavi, "Phantom: Exploiting decoder-detectable mispredictions," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 49–61. [Online]. Available: https://doi.org/10.1145/3613424.3614275

[49] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, l3 cache Side-Channel attack," in *23rd USENIX security symposium (USENIX security 14)*, 2014, pp. 719–732.

[50] H. Yavarzadeh, A. Agarwal, M. Christman, C. Garman, D. Genkin, A. Kwong, D. Moghimi, D. Stefan, K. Taram, and D. Tullsen, "Pathfinder: High-resolution control-flow attacks exploiting the conditional branch predictor," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 770–784. [Online]. Available: https://doi.org/10.1145/3620666.3651382

[51] J. Yu, L. Hsiung, M. El'Hajj, and C. W. Fletcher, "Data oblivious isa extensions for side channel-resistant and high performance computing," in *The Network and Distributed System Security Symposium (NDSS)*, 2019.

[52] J. Yu, T. Jaeger, and C. W. Fletcher, "All your pc are belong to us: Exploiting non-control-transfer instruction btb updates for dynamic pc extraction," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589100

[53] T. Zhang, K. Koltermann, and D. Evtyushkin, "Exploring branch predictors for constructing transient execution trojans," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 667–682. [Online]. Available: https://doi.org/10.1145/3373376.3378526

[54] Z. Zhang, M. Tao, S. O'Connell, C. Chuengsatiansup, D. Genkin, and Y. Yarom, "BunnyHop: Exploiting the instruction prefetcher," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 7321–7337. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/zhang-zhiyuan-bunnyhop

[55] L. Zhao, P. Li, R. Hou, M. C. Huang, X. Qian, L. Zhang, and D. Meng, "Hybp: Hybrid isolation-randomization secure branch predictor," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 346–359.

[56] Y. Zhu, B. Chen, Z. N. Zhao, and C. W. Fletcher, "Controlled preemption: Amplifying side-channel attacks from userspace," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 162–177. [Online]. Available: https://doi.org/10.1145/3676641.3715985