BranchGauge: Modeling and Quantifying Side-Channel Leakage in Randomization-Based Secure Branch Predictors

Quancheng Wang
Wuhan University
School of Cyber Science and Engineering
Wuhan, China
wangquancheng@whu.edu.cn

Ming Tang
Wuhan University
School of Cyber Science and Engineering
Wuhan, China
m.tang@whu.edu.cn

Ke Xu
Wuhan University
School of Cyber Science and Engineering
Wuhan, China
kexuwhu@whu.edu.cn

Han Wang
Wuhan University
School of Cyber Science and Engineering
Wuhan, China
han.wang@whu.edu.cn

Abstract

The inherent sharing characteristic of branch predictors makes modern processors vulnerable to microarchitectural side-channel attacks. Among proposed mitigations, randomization-based countermeasures stand out for their practical potential, offering lower performance overhead compared to flushing or partitioning techniques. However, these randomized approaches often fail to guarantee absolute security, and existing evaluation methods cannot accurately quantify leakage in such designs. This underscores the urgent need for a formal methodology to systematically model and measure side-channel leakage in randomization-based secure branch predictors. In this paper, we propose a leakage quantification framework to measure side-channel leakage in these randomized countermeasures at the microarchitecture level. Our methodology incorporates detailed modeling of the operational principles and indexing/content randomization mechanisms of PHT and BTB components, effectively capturing timing characteristics relevant to microarchitectural attacks. Based on this, we define attack strategies and secret spaces that breach security boundaries, facilitating seamless integration of microarchitectural attacks with branch predictor models. We then present leakage evaluation metrics and assess the security of existing randomized defenses under various attack strategies and secret spaces, demonstrating our framework's effectiveness in quantifying side-channel leakage. In particular, Our experiments reveal that reuse-based PHT attacks, prune-based BTB attacks and occupancy-based attacks remain significant threats, highlighting the need for stronger countermeasures.

CCS Concepts

• **Security and privacy** → **Side-channel analysis and countermeasures**; • **Hardware** → *Simulation and emulation.*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASIA CCS '25, Hanoi, Vietnam
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1410-8/25/08
https://doi.org/10.1145/3708821.3736198

Keywords

Side-Channel Attacks, Branch Predictor, Leakage Quantification

ACM Reference Format:
Quancheng Wang, Ming Tang, Ke Xu, and Han Wang. 2025. BranchGauge: Modeling and Quantifying Side-Channel Leakage in Randomization-Based Secure Branch Predictors. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS '25), August 25–29, 2025, Hanoi, Vietnam.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3708821.3736198

1 Introduction

The design philosophy of modern processors emphasizes achieving faster execution speeds and greater efficiency, with branch predictors playing a critical role in addressing control hazards. However, the sharing nature of branch predictors at the microarchitecture level exposes modern processors to various microarchitectural side-channel attacks. These attacks enable malicious attackers to exploit timing differences between correct and incorrect predictions [1–3, 6, 7, 13, 14, 17] or microarchitectural state changes induced by speculative execution [5, 19–21, 23, 27, 28, 37], thereby compromising sensitive data from other security domains.

To address the significant security risks posed by branch predictors, researchers have proposed various secure speculation designs [4, 18, 29, 40] and secure branch predictor designs [10, 22, 42–44]. Among them, randomization-based approaches such as STBPU [42] and HyBP [44] stand out as more promising compared to flush-based [9, 34] and partition-based [33] solutions. By introducing randomness to the indexing and content of branch predictors, randomization-based designs make attacks significantly more challenging, effectively reducing leakage risks while maintaining relatively low performance overhead.

Although randomization-based schemes [10, 22, 42–44] reasonably claim to enhance the security of computer systems, their effectiveness is not absolute. On the one hand, shared states between attacker and victim threads still exist, indicating that these schemes cannot ensure comprehensive security [12]. On the other hand, existing formal verification frameworks and leakage quantification methods still fall short of providing a comprehensive assessment of secure branch predictor designs. For instance, CaSA [8], Metior [12], CacheFX [15], and the framework proposed by Peters et al. [24] primarily target secure cache designs and cache replacement policies,

lacking a holistic evaluation of secure branch predictor designs. Furthermore, while Wang et al. [35, 36] analyze the security of 8 secure branch predictor designs using a symbolic modeling approach, this method is overly abstract and simplistic, making it inadequate for accurately quantifying the side-channel leakage.

To address the limitations of existing formal modeling and side-channel security evaluations for randomized secure branch predictor designs, we propose a novel leakage quantification framework. By modeling the operational principles of branch predictors and the side-channel attack methodologies targeting them, this framework enables comprehensive quantitative security analysis of randomization-based secure branch predictor designs in the context of both timing and speculative attacks.

We first analyze the fundamental working principles of randomized secure branch predictors, focusing on two critical components: Pattern History Table (PHT), which handles conditional branch prediction, and Branch Target Buffer (BTB), responsible for predicting branch target addresses. Based on the characteristics of these randomization-based designs, we propose a branch predictor model that incorporates the index randomization function (*IRF*) and the content randomization function (*CRF*). Moreover, we introduce a timing observation model to analyze the execution time of branch instructions, capturing states relevant to timing and speculative attacks. In addition, we show how existing randomized secure branch predictor designs can be integrated into this framework, facilitating further side-channel leakage evaluations of these designs.

Afterward, we systematically formalize timing and speculative attacks targeting PHT entries, BTB entries, and BTB sets, each exploiting the side-channel security vulnerabilities of branch predictors from different perspectives. These attack strategies include reuse-based attacks targeting specific PHT and BTB entries, prune-based attacks aimed at evicting BTB sets, and occupancy-based attacks that exploit the occupancy state of branch predictors. We then define the secret spaces of the victim's branch instructions to align real-world attack scenarios with our attack model, thereby enabling a comprehensive evaluation of side-channel leakage in secure branch predictor designs.

Finally, we quantitatively evaluate the side-channel leakage of existing randomized branch predictor designs using our proposed framework. We define several metrics, including the number of branch accesses, the probability of branch collisions between the attacker and the victim, and maximal leakage that can be observed. By comparing security guarantees across different attack strategies and secret spaces, we demonstrate the practicality and effectiveness of our methodology in quantifying side-channel leakage in randomization-based secure branch predictors.

Our experiments show that encryption schemes, such as Noisy-XOR-BP [43], STBPU [42] and HyBP [44], provide robust protection against reuse-based BTB attacks, thereby ensuring a high level of security in practical deployments. However, even with encrypted (e.g., Noisy-XOR-BP [43]) and wider (e.g., BSUP [22]) saturating counters, the security guarantees of existing randomized designs against PHT attacks remain insufficient. Additionally, our results highlight that prune-based and occupancy-based attacks continue to pose substantial threats, underscoring the need for more effective countermeasures. Nevertheless, we believe that our framework can be extended to serve as a valuable tool for future secure designs.

The main contributions of this paper are as follows:

- We propose a modeling methodology for evaluating side-channel leakage in randomization-based secure branch predictors at the microarchitecture level. Our approach encompasses the operational principles of branch predictors, incorporates indexing and content randomization mechanisms, and captures the execution time of branch instructions, with a focus on timing and speculative attacks.
- We formally describe the execution processes of microarchitectural attacks targeting PHT entries, BTB entries, and BTB sets. By defining reuse-based, prune-based, and occupancy-based attack strategies, along with their corresponding secret spaces, we can effectively evaluate the side-channel security properties within our framework.
- We define leakage evaluation metrics and assess the security properties of existing randomization-based secure branch predictor designs across various attack strategies and victim secret spaces. Our experiments demonstrate the effectiveness of our framework in quantifying side-channel leakage, underscoring the necessity for stronger countermeasures against microarchitectural attacks.

The research artifact of our work, including the source code and reference experimental results, is available at https://github.com/iamywang/branch-gauge.

2 Background and Related Work

In this section, we provide an overview of microarchitectural side-channel attacks targeting branch predictors, existing secure designs addressing branch predictor vulnerabilities, and modeling techniques for analyzing microarchitecture security.

2.1 Side-Channel Attacks on Branch Predictors

Depending on the attacker's strategy and the leakage model, microarchitectural attacks targeting branch predictors can be categorized into two main types: timing attacks and speculative attacks. Timing attacks represent a passive approach to information leakage, where the attacker infers sensitive data, such as encryption keys [1–3, 6, 7, 14, 17], KASLR offsets [13], or other confidential information, by analyzing the execution timing of the victim's branches.

Speculative attacks, exemplified by Spectre [20] and its numerous variants [5, 21, 23, 37], are an active form of unauthorized access. In these attacks, the attacker manipulates the branch predictor's state to influence the victim's control flow, inducing unauthorized speculative execution along incorrect paths. These exploits can undermine critical security foundations in modern computer systems, including network attack surfaces [30], memory isolation between processes [39], Trusted Execution Environments (TEEs) [11], and memory safety features such as Pointer Authentication (PAC) [28] and Memory Tagging Extension (MTE) [19].

2.2 Existing Secure Branch Prediction Designs

Secure branch predictor designs aim to address both timing and speculative attacks by introducing some hardware modifications to branch predictors. Flush-based designs, such as MI6 [9] and BRB [34], refresh branch predictor states during context switches or transitions between security domains. Partition-based designs focus

on hardware-level isolation of branch predictors to mitigate side-channel risks. For example, Lock-based BTB design [33] allocates dedicated resources for different security domains. These two types of defenses often incur significant performance overhead due to reduced branch predictor capacity.

Randomization-based secure designs employ techniques such as hardware-level non-deterministic mappings of branch instructions to predictor entries or encryption/randomization of branch predictor contents, such as BSUP [22], Noisy-XOR-BP [43], LS-BP [10], HyBP [44], and STBPU [42]. Although these designs demand significant hardware modifications, they effectively reduce the attacker's success probability while maintaining relatively low performance overhead. However, modeling and quantifying the side-channel leakage in these schemes remains an interesting research challenge.

2.3 Microarchitectural Security Evaluation

Formal verification abstracts side-channel security properties and program behaviors, leveraging reasoning and proof-based methods to ensure security guarantees. For example, Yang et al. [41] introduce Pensieve, a model checking framework for evaluating the effectiveness of hardware defenses against speculative attacks. Wang et al. [35, 36] model branch predictor components and operations for exploring attack patterns and security verification, providing a comprehensive security analysis of 8 secure branch predictor implementations against both timing and speculative attacks.

In the context of quantifying microarchitectural side-channel leakage, researchers have primarily focused on formal verification and quantification related to cache side-channels. For instance, the Prime+Prune+Probe attack method [25] and its corresponding security evaluation framework demonstrate that the interference introduced by existing randomization strategies is not as significant as expected. Techniques such as CaSA [8], Metior [12], and CacheFX [15] primarily target leakage quantification for secure cache designs. Similarly, the framework proposed by Peters et al. [24] focuses on quantifying leakage in cache replacement policies. However, these approaches do not provide a comprehensive security evaluation for randomization-based secure branch predictor designs.

3 Modeling the Structure of Security-Oriented Branch Predictors

In this section, we begin by modeling two widely used branch predictor components in modern processors that are susceptible to side-channel vulnerabilities: Pattern History Table (PHT) and Branch Target Buffer (BTB). We also describe the types of instructions considered in our branch predictor model and explain the replacement strategies employed for the BTB. Subsequently, we introduce a timing observation model to characterize the execution time of branch instructions, capturing both the hit and miss states relevant to timing attacks and the misprediction critical for speculative attacks. Finally, we demonstrate how existing randomized secure branch predictors can be implemented in this framework.

3.1 Threat Model

Microarchitectural side-channel attacks on branch predictors involve an attacker and a victim, both capable of executing branch instructions that alter the branch predictor's state. These programs may run either on the same logical processor or on separate logical processors within the same physical processor. In our threat model, the attacker and victim can operate at same privilege levels (e.g., user mode) or different ones (e.g., user mode versus kernel mode).

The victim is presumed to execute specific branch instructions tied to sensitive information, while the attacker may influence the branch predictor by executing arbitrary branch instructions. The direction of a branch depends on the secret data the attacker seeks to deduce. For instance, in the RSA encryption function implemented in the OpenSSL cryptographic library, the value of a key bit determines whether the branch is taken or not. The execution timing of these branch instructions, affected by branch predictor operations, can inadvertently reveal details about the encryption key.

Our threat model also assumes the attacker possesses some understanding of the victim's implementation, such as the cryptographic algorithm and the positions of branch instructions associated with the secret, but lacks knowledge of the actual secret data. Moreover, the attacker is assumed to be familiar with the state machine logic of the branch predictor components. While the attacker cannot directly access the internal state of the branch predictor, they can infer its state by measuring the execution time of their own or the victim's operations. By determining whether the execution is fast or slow, the attacker can deduce the branch predictor's state and infer the victim's secret data.

3.2 Defining Components and the Workflow

The development of a security-oriented branch predictor model aims to assess the security of emerging branch predictor designs and evaluate the applicability and complexity of existing side-channel attack methods against these designs. To achieve this, we introduce a branch predictor model consisting of two fundamental components: Pattern History Table (PHT) and Branch Target Buffer (BTB). This model takes a sequence of branch instructions as input and outputs the branch direction or target address for each branch.

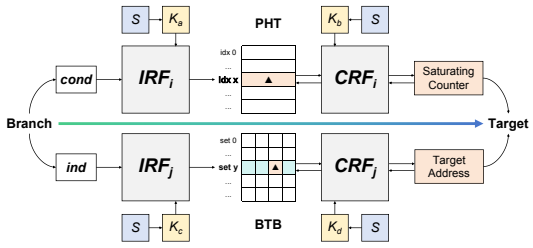

Figure 1: The workflow of our branch predictor model.

PHT Model. PHT is used to predict the direction of conditional branches (*cond*). Each PHT entry is represented by a saturating counter, with its most significant bit (MSB) determining whether the branch is predicted as "taken" or "not taken".

As shown in the upper portion of the workflow in Figure 1, our PHT model integrates two randomization functions: the index randomization function (*IRF*) and the content randomization function (*CRF*). These functions effectively reduce the attacker's success probability by introducing randomness. Both *IRF* and *CRF* use simplified inputs, including the branch instruction address or

the saturating counter content, combined with a randomization key K (e.g., process PID, XOR mask, or cryptographic key). To further enhance security, distinct keys are assigned to different security domains S, and periodic key updates are employed.

PHT Lookup. During the prediction phase of a conditional branch *cond*, the PHT determines the corresponding index by applying the *IRF* function to the branch instruction's address. Using this index, it retrieves the content of the saturating counter from the PHT. The retrieved value is then de-randomized via the *CRF* function to produce the final prediction. If the MSB of the de-randomized saturating counter is 1, the branch is predicted as "taken"; otherwise, it is predicted as "not taken".

PHT Update. Once a conditional branch instruction *cond* is executed, the PHT updates the saturating counter based on the actual branch outcome. If the branch is "taken", the saturating counter is incremented by 1; otherwise, it is decremented by 1. To further enhance randomness, the updated counter value is processed through the *CRF* function before being written back to the PHT.

BTB Model. BTB predicts the target address of indirect branches (*ind*) and is organized similarly to a set-associative cache. Each BTB entry contains several fields, including a valid bit, a tag, a target address, and replacement metadata.

Similarly, as depicted in the lower half of the workflow in Figure 1, our BTB model also incorporates two randomization mechanisms: the index randomization function (*IRF*) and the content randomization function (*CRF*). Unlike the PHT model, these functions are tailored to handle branch address-to-set mapping and entry content randomization, respectively. Additional inputs for the randomization process are abstracted using a unified key K (e.g., process PID, XOR mask, or cryptographic key), which varies across security domains S and is periodically updated for security purposes.

BTB Lookup. When an indirect branch instruction *ind* is executed, the BTB computes the set index by applying the *IRF* function to the branch instruction's address. Within the corresponding set, the BTB searches for a matching entry based on a tag comparison. If a match is found, the target address is de-randomized using the *CRF* function to generate the final prediction.

BTB Update. After executing an indirect branch instruction *ind*, the BTB updates the relevant entry with the actual target address. For added security, the target address is first randomized using the *CRF* function before being stored in the BTB. If no matching entry is found during the lookup phase, the BTB employs a replacement policy to allocate space for the new target address.

LRU Replacement. Replacement policies dictate which entry to evict when the BTB set is full and a new target address must be stored. In our methodology, we evaluate the LRU replacement policy, which prioritizes entries based on their access history. When a replacement is needed, the entry with the highest LRU metadata value—indicating the least recent access—is evicted. Each BTB entry maintains metadata tracking its LRU status. When an entry is accessed, the LRU metadata for all entries in the corresponding candidate set is updated to reflect the latest branch order.

3.3 Constructing Side-Channel Observations

Building on the characterization of the PHT and BTB components in our branch predictor model, we define a timing observation model, as illustrated in Figure 2. This model captures the timing-related states of the branch predictor and evaluates the consistency between predicted and actual outcomes.

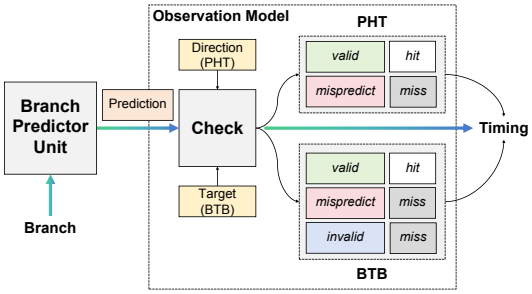

Figure 2: The side-channel timing observation in our model.

For the PHT branch predictor, if the prediction matches the actual direction (e.g., both "taken" or both "not taken"), the execution time is classified as a *hit*, and the prediction state is labeled as *valid*. If the prediction differs (e.g., one "taken" and one "not taken"), the execution time is classified as a *miss*, and the prediction state is labeled as *mispredict*. For the BTB branch predictor, if the predicted branch target matches the actual target, the execution time is classified as a *hit*, and the prediction state is labeled as *valid*. If the predicted target differs, the execution time is classified as a *miss*, and the prediction state is labeled as *mispredict*. If no matching is found in the target BTB set, the execution time is still classified as a *miss*, but the prediction state is labeled as *invalid*.

This observation model enables the analysis of timing and speculative attacks targeting branch predictors. In timing attacks, the attacker aims to distinguish timing variations between two branch execution paths. This can be achieved by observing the states generated by the model, such as differences between $(hit, valid)$ and $(miss, mispredict/invalid)$. In speculative attacks, the attacker seeks to manipulate the branch predictor's state to influence the victim's control flow. This is reflected in the state transitions between $(hit, valid)$ and $(miss, mispredict)$ produced by the model. Furthermore, the model captures the execution states and observation sequences of multiple branch instructions, enabling detailed analysis of various attack strategies and secure designs.

3.4 Implementing Existing Randomized Designs

After establishing the branch predictor workflow model and timing observation model, we integrate existing randomized secure branch predictor designs proposed by academic researchers into this framework. This instantiation process demonstrates the extensibility of our methodology for evaluating specific secure designs.

BSUP [22]. This security design generates independent private keys for each security domain, utilizes an XOR algorithm to randomize the branch predictor indices, and employs the LLBC encryption algorithm proposed in the CEASER cache [26] to randomize the contents of PHT entries and the target addresses of BTB entries. In our instantiation, we represent S and K as the attacker/victim and their corresponding randomly generated private keys, respectively, and implement the *IRF* and *CRF* functions using XOR and LLBC

encryption algorithms. Notably, the *CRF* function is applied exclusively to randomize the target addresses of PHT and BTB entries, thereby simulating the working principle of BSUP.

XOR-BP [43]. This design also generates independent private keys for attackers and victims and uses an XOR algorithm to randomize the contents of branch predictor entries, including PHT entries as well as the tags and target addresses of BTB entries. During instantiation, we associate S and K with the security domain and the corresponding private keys, respectively, and apply the *CRF* function to the PHT entries as well as the tags and target addresses of BTB entries to simulate the XOR-BP branch predictor.

Noisy-XOR-BP [43]. Building on XOR-BP, this design introduces a noise mask for branch predictor indices, which enhances the randomness observed by attackers by XORing branch addresses with an encryption scheme. During instantiation, we extend the *IRF* function as an XOR algorithm to encrypt the indices of PHT and BTB entries, thereby simulating the Noisy-XOR-BP's workflow.

LS-BP [10]. This design differs from the previous approaches by randomizing only the branch predictor indices without randomizing the contents. The core idea is to XOR the branch addresses with the process PID and then encrypt the results using PUF, achieving index randomization. In our instantiation, we set S as the process PID, define the *IRF* function as a combination of XOR and PUF encryption algorithms, and use the entire branch address as the tag for BTB entries to accurately simulate the LS-BP method.

STBPU [42]. This design generates a random number for each security domain and splits it into two parts for index randomization and content randomization. Index randomization is achieved by concatenating branch addresses with private keys and applying hash compression, while content randomization is implemented using an XOR encryption algorithm. In our instantiation, we associate S and K with the security domain and its corresponding private key, respectively. We implement the *IRF* function as a hash compression algorithm to randomize the indices of PHT entries as well as the indices and tags of BTB entries and use the *CRF* function as an XOR algorithm to randomize the contents of PHT entries and the target addresses of BTB entries, simulating the STBPU approach.

HyBP [44]. This design mitigates branch predictor side-channel attacks by partitioning and isolating small tables and randomizing large tables. Since our model focuses only on randomization designs, we model only the PHT and the last-level BTB. Specifically, we associate S and K with the security domain and the corresponding private key, implement the *IRF* function using a lightweight QARMA encryption algorithm to randomize the indices of PHT as well as the indices and tags of BTB, and apply the XOR algorithm in the *CRF* function to randomize the contents of PHT entries and the target addresses of BTB entries, simulating the HyBP scheme.

4 Fomulating Microarchitectural Side-Channel Attacks on Branch Predictors

Based on our threat model and the randomized branch predictor model, attackers can employ five poisoning methods to compromise the confidentiality of the victim's execution environment. These approaches include:

❶: Speculative attacks targeting specific PHT entries;
❷: Speculative attacks targeting specific BTB entries;
❸: Timing attacks targeting specific PHT entries;
❹: Timing attacks targeting specific BTB entries;
❺: Timing attacks targeting specific BTB sets.

In this section, we formally define these attack strategies and categorize them into reuse-based attacks, prune-based attacks, and occupancy-based attacks. Additionally, we examine typical secret spaces related to cryptographic and system security, enabling accurate simulation and characterization of real-world attack scenarios.

Since this paper focuses on the design of secure branch predictors, we assume the existence of covert channels by default in speculative attacks and use Flush+Reload—the most typical and representative covert channel—as the primary example in our attack algorithm. We also assume the use of the simplest and most commonly adopted binary (0-1) encoding for covert channels.

4.1 Reuse-Based Attacks

Figure 3 presents a high-level overview of the attack processes for reuse-based attacks, encompassing both timing and speculative variants. In line with our assumptions, we use the cache covert channel as the example for speculative attacks.

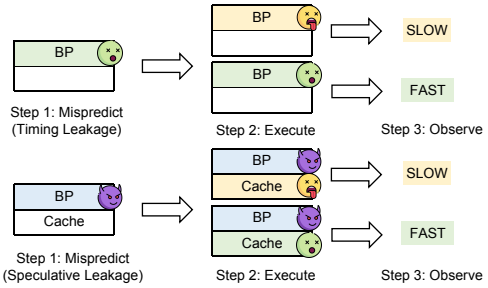

Figure 3: High-level overview of reuse-based attacks.

In Spectre V1-like attacks (❶) and PHT timing attacks (❸), the attacker exploits specific PHT entries associated with a victim branch that accesses secret data. To achieve this, the attacker crafts a conditional branch instruction whose address, after being transformed by the *IRF*, maps to the same PHT entry as the victim's branch instruction. Then, the attacker should ensure that the poisoned PHT entry results in a *mispredict* state for the victim's branch instruction, following *CRF* decoding within the victim's security domain. Such a misprediction induces timing differences for timing attacks or enables covert channel encoding (e.g., cache channels) for speculative attacks.

The attacker can iteratively evaluate a set of branch instructions to identify one that triggers the expected collision. Let $\{A_T, A_S\} \in \mathbb{A}$ represent the timing (❶) and speculative (❸) attack types, respectively, and let n denote the number of bits in the saturating counter. Define $\{D_A, D_V\} \in \mathbb{D}$ as the security domains of the attacker and victim, and $\{T_H, T_M\} \in \mathbb{T}$ as the timing observations corresponding to the *valid* and *mispredict* states, respectively. Let $\{v_{cond}\} \in \mathbb{V}$ denote the victim branch, $\{g_0, g_1, \dots\} \in \mathbb{G}$ the set of attacker's branch addresses, *cc* the covert channel address (used only in speculative attacks), and *addr* the output branch that causes the PHT collision. The detailed steps for this attack are provided in Algorithm 1.

In Spectre V2-like attacks (❷) and BTB timing attacks (❹), the attacker targets specific BTB entries. The attacker should construct an indirect branch instruction whose address, under a different *IRF* mapping, aligns with the same BTB entry as the victim's branch. The goal is to poison the BTB entry so that the victim branch mispredicts and jumps to a malicious target address. This target address, after undergoing *CRF* decoding in the victim's execution environment, is carefully chosen to induce a *mispredict* state for the victim branch in both attack types and meet the conditions for a covert channel (e.g., a valid address for cache or TLB-based channels) in speculative attacks.

The attacker can perform random tests with branch instructions to identify one that reliably triggers mispredictions for both types of attacks. For speculative attacks, the attacker must then iteratively enumerate and find a specific target address that induces the desired covert channel encoding (an additional step unnecessary in timing attacks). The detailed steps for this process are outlined in Algorithm 2. In this context, $\{A_T, A_S\} \in \mathbb{A}$ represent the timing (❷) and speculative (❹) attack types, respectively. Similarly, $\{D_A, D_V\} \in \mathbb{D}$ denote the security domains of the attacker and victim, while $\{T_H, T_M\} \in \mathbb{T}$ refer to the timing observations under the *valid* and *mispredict*/*invalid* states, respectively. The term $\{v_{ind}\} \in \mathbb{V}$ represents the victim branch, $\{g_0, g_1, \dots\} \in \mathbb{G}$ is the set of the attacker's branch addresses used for poisoning BTB tags, and $\{k_0, k_1, \dots\} \in \mathbb{K}$ denotes the set of the attacker's target addresses used for poisoning BTB contents. Additionally, *cc* signifies the target address used for the covert channel, and $\{addr, target\}$ identifies the malicious branch pair responsible for the BTB collision.

4.2 Prune-Based Attacks

Compared to reuse-based attacks, contention-based timing attacks require the attacker to construct an eviction set targeting specific entries in BTB (❺). However, due to the implementation of *IRF* randomization techniques, constructing eviction sets based on traditional branch addresses fails to cause collisions with the victim's branch addresses. Instead, it generates significant observational noise due to conflicts arising from the attacker's accesses. To overcome this challenge, Purnal et al. introduce a technique called Prime+Prune+Probe in IEEE S&P 2021 [25]. Originally designed for randomized caches, this approach can also be adapted for randomization-based branch predictors.

To apply this attack method to the BTB, the attacker first accesses a large set of branch addresses to populate the BTB (referred to as the "Prime" phase). Next, the attacker revisits the branch addresses used for the initial population and removes those that undergo self-eviction (i.e., entries marked as *invalid* and observed as *miss*). This process, known as the "Prune" phase, is repeated until no further self-evictions are detected. The attacker then triggers the victim to execute branch operations involving secret data and revisits the refined set of branch addresses (this step is referred to as the "Probe" phase). If any branches are found to be evicted due to the victim's operations, the attacker adds them to an eviction set. By iterating this process, the attacker gathers a sufficient number of colliding branches, enabling them to infer the victim's secret data. Figure 4 outlines the high-level steps of prune-based attacks, illustrating the relationship between the eviction set and the victim branch.

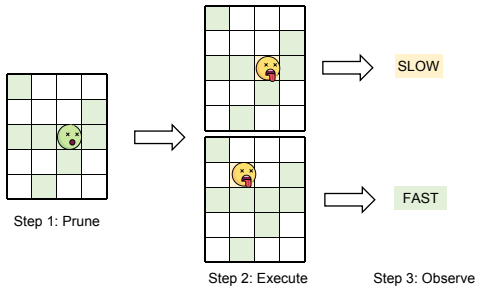

Figure 4: High-level overview of prune-based attacks.

Assume that $\{D_A, D_V\} \in \mathbb{D}$ represent the security domains of the attacker and the victim, respectively, and that $\{T_H, T_M\} \in \mathbb{T}$ correspond to timing observations associated with the *valid* and *invalid* states, respectively. Let $\{v_{ind}\} \in \mathbb{V}$ denote the victim branch, $\{k_0, k_1, \dots\} \in \mathbb{K}$ represent the set of pruning branch addresses in the attacker's domain, X denote the size of the expected eviction set, and $\{g_0, g_1, \dots\} \in \mathbb{G}$ represent the candidate eviction set. The detailed steps for this attack are outlined in Algorithm 3. To obtain a sufficiently large eviction set $\mathbb{G}$, we iteratively generate new pruning sets $\mathbb{K}$ until the size of $\mathbb{G}$ reaches the desired threshold.

4.3 Occupancy-Based Attacks

The occupancy-based attack was first proposed in USENIX Security 2019 [31], initially targeting cache contention side channels. The core idea is to infer the victim's access behavior by filling the entire cache, without knowledge of the cache's specific configuration (such as associativity, number of sets, or randomization methods). Recent studies [12, 15] demonstrate that the occupancy-based attack is a highly effective method for exploiting randomized caches. Therefore, we apply this approach to randomized branch predictors to evaluate their security properties.

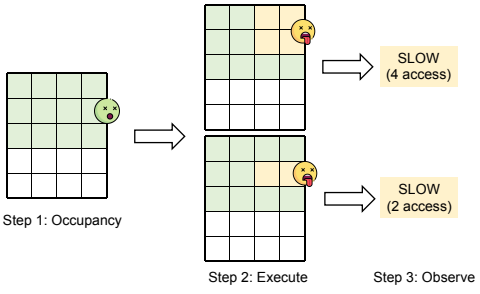

Figure 5: High-level overview of occupancy-based attacks.

In the original occupancy-based attack, the attacker fills the entire microarchitectural unit. When adapting this attack to the branch predictor, we draw inspiration from the pruning step of the Prime+Prune+Probe attack, replacing full occupancy with the construction of a set of branch addresses that do not self-conflict. The attacker then selects a specific number of branch predictor entries to occupy. After the victim's individual or set of branch instructions is executed, the attacker accesses these branch addresses

and monitors for conflicts. This approach can not only be applied to BTB-based attacks but can also be extended to PHT-based attacks. Figure 5 presents a high-level overview of the occupancy-based attack process, detailing the steps involved in filling the branch predictor and monitoring for conflicts.

The generalized steps for constructing the occupancy set are listed in Algorithm 4 and Algorithm 5. We assume that the security domains of the attacker and the victim are represented as $\{D_A, D_V\} \in \mathbb{D}$, and define $\{T_H, T_M\} \in \mathbb{T}$ to represent observations of the *valid* and *invalid/mispredict* states, respectively. Let $\{k_0, k_1, \dots\} \in \mathbb{K}$ denote the set of branch addresses controlled by the attacker, X indicate the expected size of the occupancy set, and $\{g_0, g_1, \dots\} \in \mathbb{G}$ refer to the candidate occupancy set. Once the occupancy set is constructed, the attacker can infer information about the victim by monitoring the execution of the occupancy set $\{g_0, g_1, \dots\} \in \mathbb{G}$ and the victim's branches $\{v_0, v_1, \dots\} \in \mathbb{V}$.

4.4 Victim Secret Space

We have successfully established the branch predictor execution model, the timing observation model and the attack strategies employed by attackers based on the modeling methodology. To align our analysis with real-world branch predictor attack scenarios, we now define the victim's secret space, categorized into single-bit and multi-bit secrets, as summarized in Table 1.

Table 1: Attack strategies and victim space in different attacks

Attack Type	Attack Strategy			Victim Space	
	Reuse	Prune	Occupancy	Single-Bit	Multi-Bit
❶ PHT spec	●			✓	
❷ BTB spec	●			✓	
❸ PHT entry	●		◐	✓	✓
❹ BTB entry	●			✓	✓
❺ BTB set		◐	◐	✓	✓

●: In reuse-based attacks, the attacker poisons the same branch predictor entries as the victim;
◐: In prune-based and occupancy-based attacks, the attacker creates a set of branch addresses with no self-conflicts;
✓: The victim spaces considered are indicated with a checkmark.

Single-Bit Secret. Single-bit secret spaces are commonly encountered in branch predictor attacks, particularly in cryptographic and system security domains. For instance, in cryptographic contexts, a vulnerability in the *EVP_EncryptUpdate*() function of the OpenSSL library allows attackers to extract the least significant bit (LSB) of cryptographic keys [35, 36]. In system security, attacks targeting ARM pointer authentication exploit the correctness of authentication results to compromise memory safety [28].

In this case, the victim's secret data is defined as $\mathbb{S} = \{0, 1\}$, and the branch instruction set is represented as $\{v_0, v_1\} \in \mathbb{V}$. Attackers may employ a variety of techniques, including timing attacks targeting branch predictor entries (❸ and ❹) or sets (❺) and speculative attacks (❶ and ❷). These methods exploit timing variations or speculative execution paths to infer sensitive single-bit information. Based on the assumptions in our formalized microarchitectural attack algorithm, the covert channel is limited to encoding secrets as either 0 or 1. Therefore, our analysis of speculative execution attacks considers only a single-bit secret space.

Multi-Bit Secret. Victim spaces defined by multi-bit secrets resemble threat scenarios often seen in cache attacks, which primarily exploit timing variations. However, multi-bit secrets are also pertinent in branch predictor attacks, particularly in the context of covert channel attacks and website fingerprinting. For example, variations in the victim's secret data may result in accesses to distinct branch entries in the PHT or branch sets in the BTB. In this context, the victim's secret data is expressed as $\mathbb{S} = \{0, 1\}^m$, where m represents the bit length of the secret, and the branch instruction set is denoted by $\{v_0, v_1, \dots\} \in \mathbb{V}$. Attackers can adopt strategies such as timing attacks on PHT entries (❸), BTB entries (❹) or BTB sets (❺) to infer multi-bit secrets.

5 Security Evaluation of Randomization-Based Secure Branch Predictors

In this section, we integrate the previously developed randomized secure branch predictor model with the side-channel attack model to quantify side-channel leakage in these secure designs. We first define several leakage quantification metrics and then use these metrics to compare various attack strategies and secure designs. Particular attention is given to Prime+Prune+Probe attack [15, 24, 25] and Occupancy attack [12, 15, 31], two recently proposed schemes that pose significant challenges to the security of randomization-based microarchitecture designs.

5.1 Quantification Metrics and Setup

To evaluate the effectiveness of different attack strategies and assess the security guarantees of various secure branch predictor designs, we introduce the following leakage quantification metrics:

- **Branch Accesses N:** This metric represents the total number of branch accesses required to achieve the attack's goal, encompassing accesses in both the attacker's space $\mathbb{G}$ and the victim's space $\mathbb{V}$. $\mathbf{N}$ is calculated as the following formula:

$$\mathbf{N}[\mathbb{G}, \mathbb{V}] = \sum_{i=1}^{|\mathbb{G}|} \mathbf{N}[\mathbb{G}_i] + \sum_{j=1}^{|\mathbb{V}|} \mathbf{N}[\mathbb{V}_j] \tag{1}$$

  For instance, in the context of constructing an eviction set, if 1,000 branch operations are needed to construct the eviction set for a specific branch, the value of $\mathbf{N}[\mathbb{G}, \mathbb{V}]$ would be 1,000. This type of metric has been widely used in previous security evaluations of randomized caches [15, 24], and we extend it to the security assessment of randomized branch predictors.
- **Collision Probability Pr:** This metric indicates the probability of a collision between the attacker's branches $\mathbb{G}$ and the victim's branches $\mathbb{V}$, leading to the leakage of sensitive information $s \in \mathbb{S}$ (i.e., $s = [\mathbb{V} \rightarrow \mathbb{G}]$) across $|R|$ repeated trials. This encompasses branch collisions in reuse-based attacks on individual entries, covert channel collisions in speculative attacks, and branch collisions between eviction/occupancy sets and the victim's branch in prune-based or occupancy-based

attacks. The collision probability is calculated as follows:

$$\mathbf{Pr}[s|\mathbb{G},\mathbb{V}] = \frac{1}{|R|}\sum_{i=1}^{|R|}(s = [\mathbb{V} \rightarrow \mathbb{G}]) \tag{2}$$

For example, if the attacker attempts to construct an eviction set for a specific branch address 1,000 times and succeeds 900 times, the collision probability is 0.9. This metric is also widely used in the evaluation of randomized caches [12, 24].

- **Maximal Leakage $\mathbf{L_{max}}$:** This metric quantifies the maximum amount of information that can be leaked through the attacker's space $\mathbb{G}$ and the victim's space $\mathbb{V}$ in side-channel attacks [12], while accommodating different secret spaces $\mathbb{S}$. It provides a relative measure of leakage by comparing the actual leakage to random guessing, rather than outputting an absolute leakage value in bits. The calculation formula for the maximum leakage is defined as follows:

$$\mathbf{L_{max}}[\mathbb{V} \rightarrow \mathbb{G}] = \log_2\left(\sum_{s\in\mathbb{S}} \max_{\mathbb{G}}[\mathbf{Pr}[s|\mathbb{G},\mathbb{V}]]\right) \tag{3}$$

Assume that when the victim accesses secret data 0, the attacker observes the probabilities of a branch hit and branch miss as 0.1 and 0.9, respectively, while for secret data 1, the probabilities are 0.8 and 0.2, respectively. In this case, the maximum collision probabilities are 0.9 and 0.8, respectively. The maximum leakage $\mathbf{L_{max}}$ can then be computed as $\log_2(0.9 + 0.8) = 0.77$ bits.

During the subsequent evaluation of randomized secure branch predictors, we will utilize these metrics to quantify the leakage of sensitive information across various attack strategies and victim spaces. For the branch predictor configurations, we set the PHT with 1024 entries and employ a 2-bit saturating counter (3-bit for BSUP). The BTB is configured with 1024 sets and a 4-way associativity. The maximum threshold for branch accesses is defined as $|\mathbf{N}[\mathbb{G},\mathbb{V}]| = 10^8$, based on Zhao et al. [44], who suggest that 2^{27} ($\approx 10^8$) branch accesses are infeasible in real-world attacks.

Next, to establish tight connection between the abstract branch predictor model and the formally defined microarchitectural attack methodology, we develop a software simulator with the following implementation strategy:

- **Component Structure:** In our simulator, we define vector variables to model the parameters of the PHT and BTB, explicitly specifying the counter, tag, and target address. The lookup operations for PHT and BTB entries are implemented using the *LookupPHT* and *LookupBTB* functions. The updates to PHT and BTB entries are handled through the *updatePHT* and *updateBTB* functions. These functions are designed to simulate the behavior of the PHT and BTB components according to the branch predictor model described in Section 3.2.
- **Timing Observation:** We determine timing observations based on the return values of *LookupPHT* and *LookupBTB* functions. This value indicates a *hit*(1) or a *miss*(0), aligning with the timing observation model presented in Section 3.3.
- ***IRF* and *CRF* Functions:** We compute randomized set indices for branches using the *getPHTSet* and *getBTBSet* functions. The PHT and BTB contents are obfuscated using the *getCounter*, *getBTBTag*, and *getBTBDest* functions. Each design includes its own specific function implementations, as detailed in Section 3.4.
- **Attack Algorithms:** The formally defined attack algorithms in Section 4 are implemented as simulator functions to model the attack processes and evaluate corresponding security properties. For each implementation of the attack algorithms, we collect statistics on branch access counts (i.e., the number of lookup calls) and the corresponding timing observations (used to detect branch collisions) under various attack strategies and input parameter configurations.
- **Leakage Calculation:** The statistics collected during the simulated attack process provide a solid foundation for analyzing the relationship between quantitative metrics and attacks. Based on the collected data and the previously defined formulas, we compute the **N**, **Pr**, and $\mathbf{L_{max}}$.

5.2 Reuse-Based Attack Evaluation

We first analyzing attack scenarios targeting specific entries in the branch predictor, focusing on reuse-based PHT and BTB attacks that involve both timing and speculative attacks. Using the metrics **N** and **Pr**, we evaluate the attacker's ability to create branch collisions between the attacker's branches and the victim's branches.

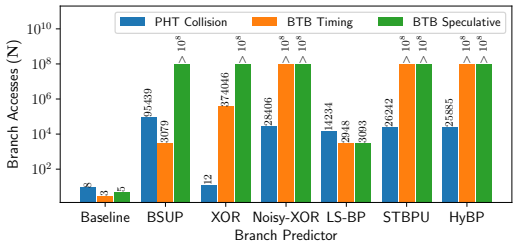

Figure 6: Number of branch accesses required for entry collision in reuse-based PHT and BTB attacks.

Evaluate Average Collison Accesses (❶–❹). As shown in Figure 6, we find that attackers can still successfully carry out PHT attacks with an average of fewer than 10^6 memory accesses under existing secure designs. In particular, for XOR-BP employing only the *CRF* content randomization, attackers can easily exploit PHTs with a limited number of saturation counter bits. In contrast, LS-BP, which only uses *IRF* index randomization, significantly increases the attack complexity. Then, security designs that integrate both randomization techniques can further enhance the attack difficulty. Additionally, we notice that BSUP, which increases the number of bits in the saturation counter based on randomization, can effectively improve PHT attack resistance.

For BTB timing attacks, BSUP and LS-BP introduce *IRF* randomization to the BTB set mapping, significantly increasing the difficulty for attackers to execute such attacks. However, attackers can still achieve collisions with an average of 3,000 branch accesses. In contrast, the XOR-BP design, which applies *CRF* randomization to BTB tags, raises the difficulty of constructing such collisions to 4×10^6 of accesses. Further analysis reveals that the security designs

Noisy-XOR-BP, STBPU, and HyBP can effectively defend against timing attacks. These designs prevent attackers from achieving malicious branch collisions involving both sets and tags, even after repeated attempts of up to 10^8 branch accesses. For speculative attacks, security designs such as BSUP, XOR-BP, Noisy-XOR-BP, STBPU, and HyBP apply *CRF* randomization to the BTB target addresses. This ensures that attackers are unable to construct malicious branches within 10^8 branch accesses.

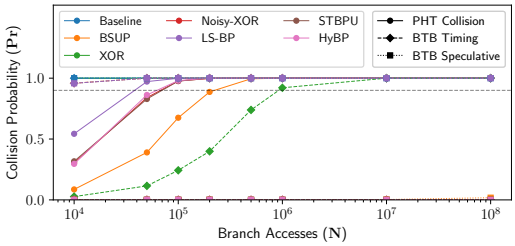

Figure 7: Collision probability for entry collision in reuse-based PHT and BTB attacks.

Calculate Collision Probability (❶–❹). We then assess the collision probability between the attacker's and the victim's branches in reuse-based PHT and BTB attacks. As illustrated in Figure 7, the collision probability increases with the number of branch accesses, denoted as **N**. For example, in the case of PHT attacks, the collision probability reaches 90% after approximately 10^5 branch accesses for all designs except BSUP, which requires about 5×10^5 accesses. Meanwhile, for BTB timing attacks, the collision probability reaches 90% after 10^4 branch accesses for the baseline, BSUP, and LS-BP designs, while XOR-BP requires about 10^6 accesses. However, for BTB speculative attacks, the 90% collision probability is rarely achieved, even after 10^8 branch accesses, except for the baseline and LS-BP designs. Additionally, we observe that Noisy-XOR-BP, STBPU, and HyBP maintain a relatively low collision probability when subjected to reuse-based BTB attacks.

Takeaways. For reuse-based attacks, *IRF* increases the attacker's complexity; however, the attacker can still generate branch conflicts within a reasonable number of branch accesses. In contrast, the security of *CRF* relies on bit width, making encrypted BTB significantly more secure than PHT. PHT reuse attacks (both timing and speculative) remain a major challenge for secure branch predictor designs.

5.3 Prune-Based Attack Evaluation

Next, we examine the Prime+Prune+Probe attack in the context of constructing eviction sets targeting particular BTB entries. Using **N** and **Pr**, we assess the attacker's capacity to create pruning sets for branch collisions and the impact of different branch accesses.

Determine Optimal BTB Pruning Set Size (❺). Previous formal verification studies on randomized caches have shown that the size of the pruning set influences the number of accesses the attacker requires to construct eviction sets. Building on this insight, we analyze how different pruning set sizes impact BTB set collisions. As depicted in Figure 8, for baseline designs and XOR-BP without indexing randomization (*IRF*), the number of branch accesses needed to create BTB set conflicts increases with the pruning set size. In contrast, the other five designs exhibit the opposite trend. For the former two designs, the optimal pruning set size, $\mathbb{K}$, is approximately 100, requiring around 400 accesses. For the latter designs, the optimal pruning set size is about 3,800, necessitating approximately 10^5 accesses.

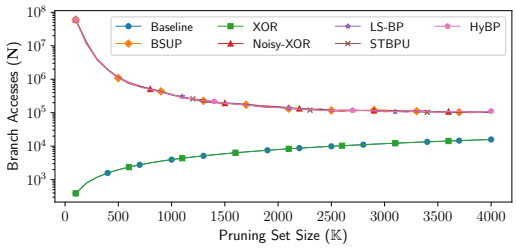

Figure 8: Number of branch accesses required for different pruning set sizes in prune-based attacks.

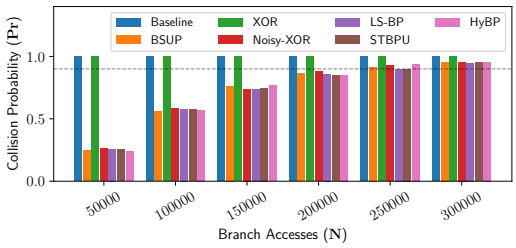

Figure 9: Collision probability for different branch accesses in prune-based attacks.

Calculate Collision Probability (❺). After determining the optimal pruning set size for different types of randomized branch predictor designs, we further analyze the impact of branch accesses on collision probability. As Figure 9 illustrates, the collision probability between the attacker's branches and the victim's branches increases with the number of branch accesses **N**. This trend is consistent across all designs, regardless of whether randomization is applied. For instance, in randomized designs with both *IRF* and *CRF*, the attacker requires approximately 2.5×10^5 branch accesses to achieve a 90% collision probability. These results indicate that even with *IRF* and *CRF* randomization, attackers can effectively leverage Prime+Prune+Probe attacks to infer the victim's secret.

Takeaways. Although *IRF* randomization is designed to prevent attackers from reasoning about the mapping between branches and entries—thereby making eviction set construction difficult—its effectiveness falls short of expectations. In practice, pruning-based strategies allow attackers to construct

eviction sets within a reasonable number of branch accesses, affecting all existing randomized BTBs.

5.4 Occupancy-Based Attack Evaluation

Subsequently, we address emerging occupancy-based attacks, which do not rely on microarchitectural parameters. Using the same metrics-**N** and **Pr**-we evaluate the size of the occupancy sets required by the attacker to fill the branch predictor, the collision probability under varying occupancy rates. In the following experiments on pruning sets and occupancy sets, we only analyze 6 secure designs and exclude the baseline. This is primarily because, for insecure PHT and BTB structures, the attacker can directly observe the mapping between branch addresses and predictor components, making occupancy-based attack strategies unnecessary.

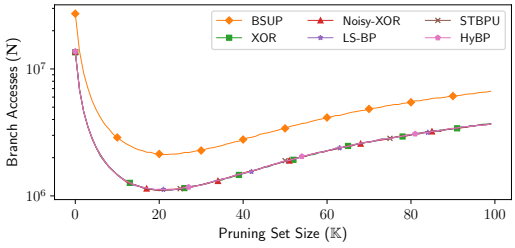

Figure 10: Number of branch accesses required for different pruning set sizes in occupancy-based PHT attacks.

Determine Optimal PHT Pruning Set Size (❹). Similar to the evaluation of prune-based attacks conducted previously, we first analyze the impact of different sizes of pruning sets on the PHT occupancy attack. The experimental results in Figure 10 indicate that as the number of branch addresses in the pruning set $\mathbb{K}$ increases, the number of branch accesses required for the attacker to fill the entire 2^{10}-entry PHT first decreases and then increases. This is because an insufficient pruning set often fails to induce adequate branch collisions, whereas large one introduces excessive self-conflicts. Both cases result in a higher number of required branch accesses. Regardless of which randomization technique is employed in PHT designs, the minimum number of branch accesses occurs when the pruning set size is about 20. For existing randomized PHTs, this value represents the security boundary and characterizes the point at which the attacker's capability is maximized. For secure designs with 2-bit saturating counters, the attacker requires approximately 10^6 branch accesses, whereas for BSUP with 3-bit saturating counters, the attacker requires around 2×10^6 branch accesses.

Determine Optimal BTB Pruning Set Size (❺). Then, we shift the focus to occupancy-based attacks targeting the BTB, beginning with an analysis of the optimal pruning set size required to execute an effective attack. As illustrated in Figure 11, when populating the entire 2^{12}-entry BTB, the number of branch accesses needed by the attacker initially decreases and then increases as the pruning set size grows. Similar to the results observed in the PHT experiments, this

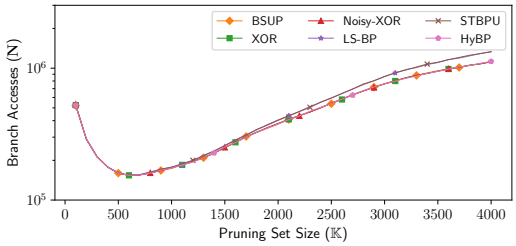

Figure 11: Number of branch accesses required for different pruning set sizes in occupancy-based BTB attacks.

trend is also caused by insufficient or excessive branch collisions. Interestingly, our findings reveal that regardless of which indexing or content randomization is employed, the attacker can still launch the attack with approximately 1.5×10^5 branch accesses (when the size of pruning set $\mathbb{K}$ is 600), even without knowledge of the BTB's exact configuration. The optimal pruning set size of 600 also reflects the security limit of existing randomized BTBs.

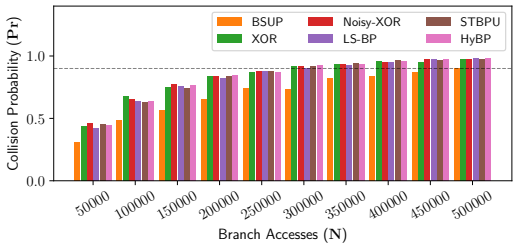

Figure 12: Collision probability for different branch accesses in occupancy-based PHT attacks.

Generate Diverse PHT Occupancy Sets (❸). Next, we evaluate the collision probability between the attacker's occupancy set and branches associated with the victim's secret data by varying the number of branch accesses. The experimental results are presented in Figure 12. As the size of the occupancy set $\mathbb{G}$ (no self-conflicts) and the number of branch accesses **N** increase, we observe a corresponding rise in the collision probability between the attacker and the victim. Specifically, the attacker needs only 3×10^5 branch accesses to achieve a 90% collision probability, with the exception of BSUP, which requires 5×10^5 branch accesses.

Generate Diverse BTB Occupancy Sets (❺). Following this, we analyze the impact of different occupancy set sizes on BTB occupancy attacks. We investigate the collision probability between the attacker and the victim through varying numbers of branch accesses. The results are illustrated in Figure 13. Similar to PHT occupancy attacks, we observe that the collision probability in BTB occupancy attacks increases with both the size of the occupancy set $\mathbb{G}$ (no self-conflicts) and the number of branch accesses **N**. Moreover, the attacker only needs to construct a BTB occupancy set with 6×10^4 branch accesses to achieve a 90% collision probability.

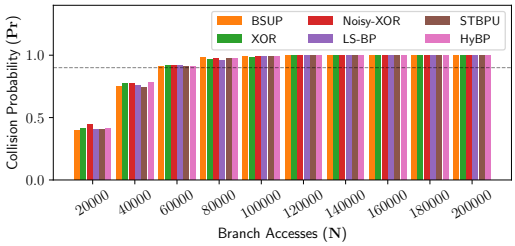

Figure 13: Collision probability for different branch accesses in occupancy-based BTB attacks.

Takeaways. When the attacker's constraints are relaxed by removing the need for knowledge of specific microarchitectural parameters and relying instead on occupancy state as the observation, the effectiveness of randomization appears to vanish. We find that despite different designs employing various random mapping or encryption functions, the uniformity of these algorithms enables attackers to easily construct occupancy sets that avoid self-conflicts.

5.5 Maximal Leakage: Single-Bit Secret Space

Furthermore, we evaluate the security of existing randomized secure branch predictors using two single-bit secret scenarios. The first scenario is like a cryptographic attack, where the secret is the LSB in OpenSSL's $EVP\_EncryptUpdate()$ function. The second scenario is like a system attack, where the sensitive data corresponds to the correctness of pointer authentication. In our evaluation, we utilize the maximal leakage metric and reuse-based attacks.

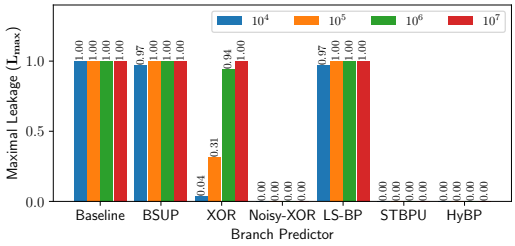

Figure 14: Maximal leakage evaluation for inferring 0/1 bits.

Inference of "0" and "1" Bits. In this experiment, the victim's secret is a randomly generated bit, either "0" or "1," which corresponds to correct and incorrect BTB predictions, respectively. The attacker-victim interaction follows these steps: the attacker first trains the BTB, then triggers the victim to execute the sensitive function, and finally infers the secret value by observing the BTB state. As shown in Figure 14, our experimental results align with previous findings: after approximately 10^6 branch accesses, the maximum leakage reaches 1.00, 0.94, and 1.00 for the BSUP, XOR-BP, and LS-BP designs, respectively. This confirms that secret data can be extracted via reuse-based BTB timing side-channel attacks. In contrast, the Noisy-XOR-BP, STBPU, and HyBP designs demonstrate relatively higher security guarantees.

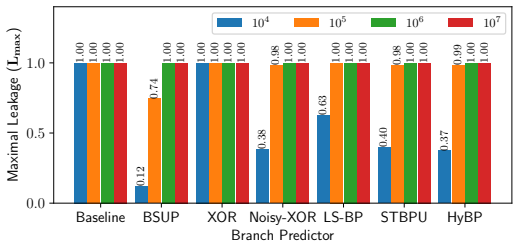

Figure 15: Maximal leakage evaluation for inferring speculative execution paths.

Inference of Speculative Execution Paths. In this experiment, we assume that a covert channel is always present, and different speculative execution paths encode distinct secret values through the covert channel during PHT speculative execution. The attacker-victim interaction follows these steps: the attacker first trains the PHT, then triggers the victim's speculative execution, and finally recovers the secret data by analyzing the covert channel state. The experimental results illustrated in Figure 15 corroborate prior reuse attack analyses: regardless of whether a 2-bit or 3-bit saturating counter is used, the maximum leakage reaches 1.00 after approximately 10^6 branch accesses. This demonstrates that attackers can effectively exploit PHT-based speculative execution attacks to recover secret data from the victim.

5.6 Maximal Leakage: Multi-Bit Secret Space

Previous experiments focus on collisions between the attacker's and victim's branches within a single-bit secret space. In this work, we extend our study to multi-bit secret spaces and explore covert channel attacks based on occupancy strategies as case studies.

Table 2: Maximal leakage for different attacker's branch accesses and number of iterations in PHT occupancy attacks

Design	1 Iteration			2 Iterations			4 Iterations		
	10^4	10^5	5×10^5	10^4	10^5	5×10^5	10^4	10^5	5×10^5
BSUP	0.01	0.01	0.39	0.08	0.14	0.89	0.29	0.64	1.56
XOR	0.00	0.00	0.66	0.10	0.26	1.15	0.39	0.90	1.81
Noisy-XOR	0.00	0.00	0.66	0.11	0.27	1.15	0.38	0.90	1.81
LS-BP	0.01	0.01	0.66	0.10	0.27	1.15	0.37	0.91	1.81
STBPU	0.00	0.00	0.65	0.11	0.25	1.14	0.38	0.89	1.80
HyBP	0.00	0.00	0.66	0.12	0.25	1.15	0.39	0.88	1.81

PHT Leakage Evaluation (❸). We first evaluate the maximum leakage of existing randomized secure branch predictors under PHT attacks. This leakage is determined by the probability of branch conflicts observed by the attacker, the victim's branch access behavior, and the number of iterations performed by the attacker. As shown in Table 2, the evaluation results indicate that designs using 2-bit saturating counters exhibit similar performance metrics. The

maximum leakage increases as the number of branch accesses by the attacker grows, reaching approximately 0.66 when the attacker performs 5×10^5 accesses. The BSUP design, which employs 3-bit saturating counters, demonstrates relatively lower leakage, with a maximum value of approximately 0.39. Moreover, as the number of iterations by the attacker increases, the maximum leakage also rises, and the differences between saturating counters of different sizes gradually diminish. When the attacker iterates four times, the maximum leakage reaches approximately 1.81 for the 2-bit saturating counter and 1.56 for the BSUP, demonstrating the inadequacy of current randomization strategies against PHT attacks.

Table 3: Maximal leakage for different attacker's branch accesses and number of iterations in BTB occupancy attacks

Design	1 Iteration			2 Iterations			4 Iterations		
	10^4	10^5	2×10^5	10^4	10^5	2×10^5	10^4	10^5	2×10^5
BSUP	0.00	0.71	0.74	0.09	1.19	1.22	0.37	1.85	1.87
XOR	0.01	0.71	0.74	0.11	1.19	1.22	0.39	1.85	1.87
Noisy-XOR	0.01	0.70	0.74	0.10	1.19	1.22	0.38	1.84	1.87
LS-BP	0.01	0.71	0.74	0.11	1.19	1.22	0.39	1.85	1.87
STBPU	0.00	0.70	0.74	0.09	1.19	1.22	0.38	1.84	1.87
HyBP	0.01	0.70	0.74	0.10	1.19	1.22	0.39	1.84	1.87

BTB Leakage Evaluation (❺). Afterward, we evaluate the maximum leakage of existing randomized secure branch predictors under BTB attacks. The leakage is also determined by the branch conflict probabilities observed by the attacker, the victim's branch access behavior, and the number of iterations performed by the attacker. As demonstrated in Table 3, the leakage behavior remains consistent regardless of whether randomization mechanisms are applied, including index randomization, content randomization, or a combination of both, when subjected to contention-based attacks where the attacker ignores these parameter details. This observation aligns with the results of previous experiments. In scenarios where a single iteration involves up to 2×10^5 branch accesses, the measured maximum leakage is approximately 0.74. However, when the attacker increases the iterations to two, the maximum leakage rises to about 1.22, and further increases to four iterations result in a maximum leakage of approximately 1.87, highlighting the limitations of current randomization designs against BTB attacks.

Takeaways. After the systematic evaluation of existing randomized secure branch predictors, we conclude that their security guarantees remain limited. While Noisy-XOR-BP, STBPU, and HyBP effectively mitigate BTB reuse attacks, all existing designs remain vulnerable to other attack vectors, including PHT reuse attacks, BTB pruning attacks, and PHT/BTB occupancy attacks.

6 Discussion

Comparison with Prior Work. Compared to existing microarchitectural evaluation frameworks, our work introduces innovations in the following aspects. First, in terms of the target of leakage quantification frameworks, most existing works focus on randomized cache structure designs (e.g., CaSA [8], Prime+Prune+Probe attack [25], Metior [12], and CacheFX [15]) or different cache replacement policies (e.g., the framework by Peters et al. [24]). In contrast, our work shifts the focus to branch predictor structures, which have received comparatively less attention in this context despite playing a critical role in microarchitectural security. Second, regarding leakage quantification metrics, our work effectively integrates the strengths of state-of-the-art frameworks such as Metior [12] and CacheFX [15]. Unlike approaches that rely solely on mutual information or access entropy, our framework can simultaneously evaluate attack complexity, collision probability, and maximum leakage, thereby enabling more interpretable and comprehensive leakage quantification [38]. Third, with respect to branch prediction, although the works by Wang et al. [35, 36] also evaluate the security guarantees of secure branch predictor designs, their three-step attack model is overly simplified and limited to qualitative analysis. In comparison, our model of branch predictors and formalized attack algorithms captures fine-grained characteristics of branch accesses, allowing for precise, quantitative evaluation of side-channel leakage.

Future Directions and Extensibility. Future research should prioritize on developing more effective countermeasures, such as hybrid designs like SassCache [16] or non-deterministic approaches like PhantomCache [32] in the cache domain. Meanwhile, our framework can be easily extended to assess such designs. Similar to the approach taken by Giner et al. [16], who integrated SassCache into CacheFX, the variable parameters and function implementations in our framework can be adapted to support the security analysis of partitioning-based and more complex designs, beyond just randomized ones. Additionally, more attention should be given to the PHT, which remains the weakest link in current secure designs and is particularly vulnerable to Spectre attacks.

7 Conclusion

This paper introduces a leakage quantification framework for modeling microarchitectural attacks and quantifying side-channel leakage in randomization-based secure branch predictors during the early design phase. We first develop a branch predictor model focused on side-channel security, emphasizing the PHT and BTB components, integrating index and content randomization techniques, and incorporating timing observation mechanisms to capture both timing and speculative attacks. Next, we formalize different types of microarchitectural attack strategies and victim secret spaces, based on the characteristics of existing branch predictor vulnerabilities. Finally, we introduce quantification metrics from the perspectives of attacker cost and information leakage, using them to assess the side-channel security properties of randomized branch predictor designs. While our experiments show that secure branch predictor designs can mitigate parts of security threats, existing countermeasures fail to effectively mitigate reuse-based PHT attacks, prune-based and occupancy-based attacks, underscoring the need for further development of secure branch predictors.

Acknowledgments

This work was supported by the National Key R&D Program of China under Grant No. 2022YFB3103800. We would also like to

thank the shepherd and anonymous reviewers of AsiaCCS 2025 for their constructive and insightful comments.

References

[1] Onur Acıiçmez, Shay Gueron, and Jean-Pierre Seifert. 2007. New Branch Prediction Vulnerabilities in OpenSSL and Necessary Software Countermeasures. In *Proceedings of the 11th IMA International Conference on Cryptography and Coding*. Springer, 185–203.

[2] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2007. On the Power of Simple Branch Prediction Analysis. In *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security*. ACM, 312–320.

[3] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2007. Predicting Secret Keys via Branch Prediction. In *Proceedings of the 7th Cryptographers' track at the RSA conference on Topics in Cryptology*. Springer, 225–242.

[4] Sam Ainsworth and Timothy M Jones. 2020. MuonTrap: Preventing Cross-Domain Spectre-Like Attacks by Capturing Speculative State. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 132–144.

[5] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. 2022. Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, 971–988.

[6] Sarani Bhattacharya, Clémentine Maurice, Shivam Bhasin, and Debdeep Mukhopadhyay. 2017. Template Attack on Blinded Scalar Multiplication with Asynchronous perf-ioctl Calls. *IACR Cryptology ePrint Archive* (2017), 968.

[7] Sarani Bhattacharya, Clémentine Maurice, Shivam Bhasin, and Debdeep Mukhopadhyay. 2019. Branch Prediction Attack on Blinded Scalar Multiplication. *IEEE Trans. Comput.* 69, 5 (2019), 633–648.

[8] Thomas Bourgeat, Jules Drean, Yuheng Yang, Lillian Tsai, Joel Emer, and Mengjia Yan. 2020. CaSA: End-to-end Quantitative Security Analysis of Randomly Mapped Caches. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1110–1123.

[9] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, and Srinivas Devadas. 2019. MI6: Secure Enclaves in a Speculative Out-of-Order Processor. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 42–56.

[10] Congcong Chen, Chaoqun Shen, and Jiliang Zhang. 2022. Lightweight and Secure Branch Predictors against Spectre Attacks. In *2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 25–30.

[11] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. 2019. SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 142–157.

[12] Peter W Deutsch, Weon Taek Na, Thomas Bourgeat, Joel S Emer, and Mengjia Yan. 2023. Metior: A Comprehensive Model to Evaluate Obfuscating Side-Channel Defense Schemes. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. ACM, 1–16.

[13] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–13.

[14] Dmitry Evtyushkin, Ryan Riley, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2018. 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*. ACM, 693–707.

[15] Daniel Genkin, William Kosasih, Fangfei Liu, Anna Trikalinou, Thomas Unterluggauer, and Yuval Yarom. 2023. CacheFX: A Framework for Evaluating Cache Security. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*. ACM, 163–176.

[16] Lukas Giner, Stefan Steinegger, Antoon Purnal, Maria Eichlseder, Thomas Unterluggauer, Stefan Mangard, and Daniel Gruss. 2023. Scatter and Split Securely: Defeating Cache Contention and Occupancy Attacks. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2273–2287.

[17] Tianlin Huo, Xiaoni Meng, Wenhao Wang, Chunliang Hao, Pei Zhao, Jian Zhai, and Mingshu Li. 2020. Bluethunder: A 2-level Directional Predictor Based Side-Channel Attack against SGX. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020, 1 (2020), 321–347.

[18] Khaled N Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2019. SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[19] Juhee Kim, Jinbum Park, Sihyeon Roh, Jaeyoung Chung, Youngjoo Lee, Taesoo Kim, and Byoungyoung Lee. 2025. TikTag: Breaking ARM's Memory Tagging Extension with Speculative Execution. In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE.

[20] 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. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1–19.

[21] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*. 1–12.

[22] Jaekyu Lee, Yasuo Ishii, and Dam Sunwoo. 2020. Securing Branch Predictors with Two-Level Encryption. *ACM Transactions on Architecture and Code Optimization* 17, 3 (2020), 1–25.

[23] Giorgi Maisuradze and Christian Rossow. 2018. ret2spec: Speculative Execution Using Return Stack Buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2109–2122.

[24] Moritz Peters, Nicolas Gaudin, Jan Philipp Thoma, Vianney Lapôtre, Pascal Cotret, Guy Gogniat, and Tim Güneysu. 2024. On The Effect of Replacement Policies on The Security of Randomized Cache Architectures. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*. ACM, 483–497.

[25] Antoon Purnal, Lukas Giner, Daniel Gruss, and Ingrid Verbauwhede. 2021. Systematic Analysis of Randomization-based Protected Cache Architectures. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 987–1002.

[26] Moinuddin K Qureshi. 2018. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 775–787.

[27] Hany Ragab, Andrea Mambretti, Anil Kurmus, and Cristiano Giuffrida. 2024. GhostRace: Exploiting and Mitigating Speculative Race Conditions. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, 6185–6202.

[28] Joseph Ravichandran, Weon Taek Na, Jay Lang, and Mengjia Yan. 2022. PACMAN: attacking ARM pointer authentication with speculative execution. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. ACM, 685–698.

[29] Gururaj Saileshwar and Moinuddin K Qureshi. 2019. CleanupSpec: An "Undo" Approach to Safe Speculation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 73–86.

[30] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. 2019. NetSpectre: Read Arbitrary Memory over Network. In *24th European Symposium on Research in Computer Security*. Springer, 279–299.

[31] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. 2019. Robust Website Fingerprinting Through the Cache Occupancy Channel. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, 639–656.

[32] Qinhan Tan, Zhihua Zeng, Kai Bu, and Kui Ren. 2020. PhantomCache: Obfuscating Cache Conflicts with Localized Randomization. In *Network and Distributed Systems Security (NDSS) Symposium 2020*. ISOC, 1–17.

[33] Ya Tan, Jizeng Wei, and Wei Guo. 2014. The Micro-architectural Support Countermeasures against the Branch Prediction Analysis Attack. In *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 276–283.

[34] Ilias Vougioukas, Nikos Nikoleris, Andreas Sandberg, Stephan Diestelhorst, Bashir M Al-Hashimi, and Geoff V Merrett. 2019. BRB: Mitigating Branch Predictor Side-Channels. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 466–477.

[35] Quancheng Wang, Ming Tang, Ke Xu, and Han Wang. 2024. Modeling, Derivation, and Automated Analysis of Branch Predictor Security Vulnerabilities. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 409–423.

[36] Quancheng Wang, Ming Tang, Ke Xu, and Han Wang. 2025. Unveiling and Evaluating Vulnerabilities in Branch Predictors via a Three-Step Modeling Methodology. *ACM Transactions on Architecture and Code Optimization* 22, 1 (2025), 1–26.

[37] Johannes Wikner and Kaveh Razavi. 2022. RETBLEED: Arbitrary Speculative Code Execution with Return Instructions. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, 3825–3842.

[38] Benjamin Wu, Aaron B Wagner, and G Edward Suh. 2020. A Case for Maximal Leakage as A Side Channel Leakage Metric. *arXiv preprint arXiv:2004.08035* (2020), 1–21.

[39] Wenjie Xiong and Jakub Szefer. 2021. Survey of Transient Execution Attacks and Their Mitigations. *Comput. Surveys* 54, 3 (2021), 1–36.

[40] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. 2018. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 428–441.

[41] Yuheng Yang, Thomas Bourgeat, Stella Lau, and Mengjia Yan. 2023. Pensieve: Microarchitectural Modeling for Security Evaluation. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. ACM, 1–15.

[42] Tao Zhang, Timothy Lesch, Kenneth Koltermann, and Dmitry Evtyushkin. 2022. STBPU: A Reasonably Secure Branch Prediction Unit. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 109–123.

[43] Lutan Zhao, Peinan Li, Rui Hou, Michael C Huang, Jiazhen Li, Lixin Zhang, Xuehai Qian, and Dan Meng. 2021. A Lightweight Isolation Mechanism for Secure Branch Predictors. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1267–1272.

[44] Lutan Zhao, Peinan Li, Rui Hou, Michael C Huang, Xuehai Qian, Lixin Zhang, and Dan Meng. 2022. HyBP: Hybrid Isolation-Randomization Secure Branch Predictor. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 346–359.

A Formalized Microarchitectural Attack Algorithms on Branch Predictors

Algorithm 1: Reuse-Based Attacks on PHT. The attacker can iteratively evaluate a set of branch instructions to identify one that triggers the expected collision. Let $\{A_T, A_S\} \in \mathbb{A}$ represent the timing (❶) and speculative (❸) attack types, respectively, and let n denote the number of bits in the saturating counter. Define $\{D_A, D_V\} \in \mathbb{D}$ as the security domains of the attacker and victim, and $\{T_H, T_M\} \in \mathbb{T}$ as the timing observations corresponding to the *valid* and *mispredict* states, respectively. Let $\{v_{cond}\} \in \mathbb{V}$ denote the victim branch, $\{g_0, g_1, \dots\} \in \mathbb{G}$ the set of attacker's branch addresses, *cc* the covert channel address (used only in speculative attacks), and *addr* the output branch that causes the PHT collision. The detailed steps for this attack are provided in Algorithm 1.

Algorithm 1: Steps for reuse-based attacks on PHT entries

Input: Attack type $\{A_T, A_S\} \in \mathbb{A}$; bits in saturating counter n; Security domains $\{D_A, D_V\} \in \mathbb{D}$; Timing observations $\{T_H, T_M\} \in \mathbb{T}$; Victim branch $\{v_{cond}\} \in \mathbb{V}$; Set of attacker's branches $\{g_0, g_1, \dots\} \in \mathbb{G}$; Covert channel address *cc*.

Output: Malicious branch address *addr*.

```
for g ∈ 𝔾 do
    for ctr ∈ {0,1}^n do
        // train PHT entries;
        for i ∈ ctr do
            PHT.lookup(g, D_A);
        // trigger victim branch and check states;
        if A_T then
            assign T ← PHT.lookup(v_cond, D_V);
            if T = T_M then
                assign addr ← g;
                return;
        else if A_S then
            PHT.lookup(v_cond, D_V);
            assign T ← access(cc);
            if T = T_H then
                assign addr ← g;
                return;
```

Algorithm 2: Reuse-Based Attacks on BTB. The attacker can perform random tests with branch instructions to identify one that reliably triggers mispredictions for both types of attacks. For speculative attacks, the attacker must then iteratively enumerate and find a specific target address that induces the desired covert channel encoding (an additional step unnecessary in timing attacks). The detailed steps for this process are outlined in Algorithm 2. In this context, $\{A_T, A_S\} \in \mathbb{A}$ represent the timing (❷) and speculative (❹) attack types, respectively. Similarly, $\{D_A, D_V\} \in \mathbb{D}$ denote the security domains of the attacker and victim, while $\{T_H, T_M\} \in \mathbb{T}$ refer to the timing observations under the *valid* and *mispredict*/*invalid* states, respectively. The term $\{v_{ind}\} \in \mathbb{V}$ represents the victim branch, $\{g_0, g_1, \dots\} \in \mathbb{G}$ is the set of the attacker's branch addresses used for poisoning BTB tags, and $\{k_0, k_1, \dots\} \in \mathbb{K}$ denotes the set of the attacker's target addresses used for poisoning BTB contents. Additionally, *cc* signifies the target address used for the covert channel, and $\{addr, target\}$ identifies the malicious branch pair responsible for the BTB collision.

Algorithm 2: Steps for reuse-based attacks on BTB entries

Input: Attack type $\{A_T, A_S\} \in \mathbb{A}$; Security domains $\{D_A, D_V\} \in \mathbb{D}$; Timing observations $\{T_H, T_M\} \in \mathbb{T}$; Victim branch $\{v_{ind}\} \in \mathbb{V}$; Set of attacker's branches $\{g_0, g_1, \dots\} \in \mathbb{G}$; Set of attacker's targets $\{k_0, k_1, \dots\} \in \mathbb{K}$; Covert channel address *cc*.

Output: Malicious pair of branch addresses $\{addr, target\}$.

```
for g ∈ 𝔾 do
    // train BTB entries;
    assign k ← k_0;
    assign p ← {g, k};
    BTB.lookup(p, D_A);
    // trigger victim branch and check states;
    assign T ← BTB.lookup(v_ind, D_V);
    if T = T_M then
        if A_T then
            assign {addr, target} ← p;
            return;
        else if A_S then
            for k' ∈ 𝕂 do
                assign p' ← {g, k'};
                BTB.lookup(p', D_A);
                BTB.lookup(v_ind, D_V);
                assign T' ← access(cc);
                if T' = T_H then
                    assign {addr, target} ← p';
                    return;
```

Algorithm 3: Prune-Based Attacks. Assume that $\{D_A, D_V\} \in \mathbb{D}$ represent the security domains of the attacker and the victim, respectively, and that $\{T_H, T_M\} \in \mathbb{T}$ correspond to timing observations associated with the *valid* and *invalid* states, respectively. Let $\{v_{ind}\} \in \mathbb{V}$ denote the victim branch, $\{k_0, k_1, \dots\} \in \mathbb{K}$ represent the set of pruning branch addresses in the attacker's domain, X denote the size of the expected eviction set, and $\{g_0, g_1, \dots\} \in \mathbb{G}$ represent the candidate eviction set. To obtain a sufficiently large eviction set $\mathbb{G}$, we iteratively generate new pruning sets $\mathbb{K}$ until the size of $\mathbb{G}$ reaches the desired threshold.

Algorithm 4: Steps for constructing occupancy set on PHT

Input: Security domains $\{D_A, D_V\} \in \mathbb{D}$; Timing observations $\{T_H, T_M\} \in \mathbb{T}$; Pruning set $\{k_0, k_1, \dots\} \in \mathbb{K}$; Size of occupancy set X.
Output: Occupancy set $\{g_0, g_1, \dots\} \in \mathbb{G}$.

```
while |𝔾| < X do
    do
        // prime and prune self-conflicts;
        for k ∈ 𝕂 do
            for k′ ∈ 𝕂 \ {k} do
                assign T ← checkConflict(k, k′);
                if T = T_M then
                    assign 𝕂 ← 𝕂 \ {k};
    while self-eviction detected;
    // probe collision with 𝔾;
    for g ∈ 𝔾 do
        PHT.lookup(g, D_A);
    for k ∈ 𝕂 do
        assign T′ ← PHT.lookup(k, D_A);
        if T′ = T_H then
            assign 𝔾 ← 𝔾 ∪ {k};
```

Algorithm 5: Steps for constructing occupancy set on BTB

Input: Security domains $\{D_A, D_V\} \in \mathbb{D}$; Timing observations $\{T_H, T_M\} \in \mathbb{T}$; Pruning set $\{k_0, k_1, \dots\} \in \mathbb{K}$; Size of occupancy set X.
Output: Occupancy set $\{g_0, g_1, \dots\} \in \mathbb{G}$.

```
while |𝔾| < X do
    do
        // prime and prune self-conflicts;
        for k ∈ 𝕂 do
            BTB.lookup(k, D_A);
        for k ∈ 𝕂 do
            assign T ← BTB.lookup(k, D_A);
            if T = T_M then
                assign 𝕂 ← 𝕂 \ {k};
    while self-eviction detected;
    // probe collision with 𝔾;
    for g ∈ 𝔾 do
        BTB.lookup(g, D_A);
    for k ∈ 𝕂 do
        assign T′ ← BTB.lookup(k, D_A);
        if T′ = T_H then
            assign 𝔾 ← 𝔾 ∪ {k};
```

Algorithm 3: Steps for prune-based attacks on BTB sets

Input: Security domains $\{D_A, D_V\} \in \mathbb{D}$; Timing observations $\{T_H, T_M\} \in \mathbb{T}$; Victim branch $\{v_{ind}\} \in \mathbb{V}$; Pruning set $\{k_0, k_1, \dots\} \in \mathbb{K}$; Size of eviction set X.
Output: Eviction set $\{g_0, g_1, \dots\} \in \mathbb{G}$.

```
while |𝔾| < X do
    do
        // prime and prune self-evictions;
        for k ∈ 𝕂 do
            BTB.lookup(k, D_A);
        for k ∈ 𝕂 do
            assign T ← BTB.lookup(k, D_A);
            if T = T_M then
                assign 𝕂 ← 𝕂 \ {k};
    while self-eviction detected;
    // probe collision with victim branch;
    BTB.lookup(v_ind, D_V);
    for k ∈ 𝕂 do
        assign T′ ← BTB.lookup(k, D_A);
        if T′ = T_M then
            assign 𝔾 ← 𝔾 ∪ {k};
```

Algorithm 4 and 5: Occupancy-Based Attacks. The generalized steps for constructing the occupancy set are listed in Algorithm 4 and Algorithm 5. We assume that the security domains of the attacker and the victim are represented as $\{D_A, D_V\} \in \mathbb{D}$, and define $\{T_H, T_M\} \in \mathbb{T}$ to represent observations of the *valid* and *invalid*/*mispredict* states, respectively. Let $\{k_0, k_1, \dots\} \in \mathbb{K}$ denote the set of branch addresses controlled by the attacker, X indicate the expected size of the occupancy set, and $\{g_0, g_1, \dots\} \in \mathbb{G}$ refer to the candidate occupancy set. Once the occupancy set is constructed, the attacker can infer information about the victim by monitoring the execution of the occupancy set $\{g_0, g_1, \dots\} \in \mathbb{G}$ and the victim's branches $\{v_0, v_1, \dots\} \in \mathbb{V}$.