CryptoBTB: A Secure Hierarchical BTB for Diverse Instruction Footprint Workloads

Debpratim Adak
North Carolina State University
Raleigh, USA
dadak@ncsu.edu

Eric Rotenberg
North Carolina State University
Raleigh, USA
ericro@ncsu.edu

Amro Awad
University of Oxford
Oxford, United Kingdom
amro.awad@eng.ox.ac.uk

Huiyang Zhou
North Carolina State University
Raleigh, USA
hzhou@ncsu.edu

Abstract

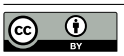
Timing attacks leveraging shared resources on a CPU are a growing concern. Branch Target Buffer (BTB), a crucial component of high-performance processors, is shared among threads and privileged spaces. Recently, researchers discovered numerous vulnerabilities in the BTB, allowing an adversary to maliciously infer a victim's BTB update and mistrain the BTB. Such attacks can successfully bypass privilege-level and secure enclave protection, as well as address space isolation. Randomizing BTB through encrypted addressing to prevent these attacks suffers from high performance overhead due to exposed encryption latency in the pipeline. Prior works address this by using encryption schemes that are either not fully secure or require frequent flush. The most recent proposal, HyBP [79], uses stronger encryption schemes. However, it suffers from high overhead since it underutilizes the BTB and suffers from higher collisions within the same thread.

In this work, we propose CryptoBTB, a secure BTB, specifically designed for an exclusive BTB hierarchy. Our proposal decouples the index encryption from the index itself, enabling low-latency index encryption to obscure BTB set mapping. Additionally, unlike earlier secure BTB proposals, this scheme is well-suited for applications with a higher instruction footprint where performance is sensitive to the BTB capacity. We evaluated CryptoBTB on various classes of workloads that stress the BTB differently. Our results show that CryptoBTB incurs 4.27% performance overhead for these workloads, while HyBP experiences 31.89% overhead. Moreover, CryptoBTB requires 22.57% lower hardware overhead than HyBP.

Keywords

Branch Target Buffer, Side-Channel

ACM Reference Format:
Debpratim Adak, Eric Rotenberg, Amro Awad, and Huiyang Zhou. 2025. CryptoBTB: A Secure Hierarchical BTB for Diverse Instruction Footprint Workloads. In *58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25), October 18–22, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3725843.3756050



MICRO '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-1573-0/25/10
https://doi.org/10.1145/3725843.3756050

1 Introduction

In modern processors, techniques such as Simultaneous Multithreading (SMT) and multitasking through context switches enable efficient resource sharing between threads and privilege levels to maximize resource utilization. However, shared resources present potential sources of side channels that can be abused by potential adversaries. For example, speculative execution attacks often use shared resources to leave a remnant footprint. Recently, researchers showed how a shared Branch Target Buffer (BTB) and/or cache hierarchy enable various exploits that can potentially leak secret keys [12, 13, 26], disclose kernel memory bypassing privilege protection[44, 50], and de-randomized KASLR protection [32, 77], etc.

We classify attacks targeting shared BTB into two categories: (a) conflict-based attacks, which are enabled by conflicts within a BTB set [12, 13] and (b) collision-based attacks, resulting from false positives due to partial virtual tag checking [32, 71, 73, 77]. To protect against these attacks, solutions adopted in recent CPU generations include Indirect Branch Restricted Speculation (IBRS) [6], Single Thread Indirect Branch Predictors (STIBP) [9], and Indirect Branch Predictor Barrier (IBPB) [5], and suppressing branch prediction on non-branch (suppressBPOnNonBr)[71]. These solutions could protect only a subset of BTB vulnerabilities. The state-of-the-art academic proposals to secure BTB require encrypting both the BTB index and content to prevent conflict-based and collision-based attacks, respectively [35, 48, 76, 78, 79]. The majority of the proposals employ either static [35, 48, 78] or non-standard single cycle encryption primitives [76] to eliminate the costly encryption latency of standard encryption algorithms. While static encryption requires frequent key (key used for encrypting the index) updates to protect against conflict-based attacks, the use of a non-standard encryption algorithm poses a considerable security threat. The most recent solution, HyBP[79], lengthens the flush period by using a novel low-latency but standard encryption primitive. However, it incurs high overhead due to a higher instruction misfetch caused by an increased collision in the encrypted indices. In this case, multiple PCs with the same BTB tag but distinct original indices are mapped to the same BTB set after index encryption. In addition, this design suffers from BTB underutilization. As a result, HyBP has 47.44% overhead on average for server workloads where the performance is limited by the BTB capacity [25, 46, 54, 64]. Workloads from the SPEC2017 suite experience a 9.11% slowdown on average in

this architecture, even though these workloads do not require a large BTB to accommodate instruction working sets, unlike server workloads.

To overcome the limitations of existing proposals, we propose CryptoBTB, an efficient and secure BTB hierarchy that is suitable for workloads with diverse instruction footprints. The CryptoBTB design randomizes address-to-set mapping to protect against conflict-based attacks without incurring the drawbacks of prior secure BTB proposals.

CryptoBTB protects against conflict-based attacks using address-to-set randomization to obscure the BTB set mapping, thereby complicating the attacker's task of identifying the eviction set. The eviction set refers to the smallest number of PC groups that can map and subsequently evict the victim's address. This scheme is prevalent in the L3cache randomization for conflict-based attack prevention, where most of these academic proposals use low-latency block ciphers such as PRINCE [27] and QARMA [17] to randomize the index [61, 70]. However, adopting such solutions for the BTB is unsuitable because of the exposed encryption latency in the pipeline. To reduce encryption latency while still achieving index randomization, we separate the index encryption and decryption from the index itself using cryptographic pads, which are a set of random numbers generated using the block cipher. This approach allows us to cache the pads with high temporal locality since CryptoBTB uses the same pad for a region. The region is derived by partitioning the virtual address space equally. Thus, we achieve a low latency encryption for a hit in this cache. This encryption technique mitigates all the drawbacks observed in HyBP.

Unfortunately, our analysis reveals that frequent BTB remapping is necessary due to its small size to prevent the adversary from determining the eviction set. This is detrimental to the BTB hierarchy because CryptoBTB does not immediately reinstall existing BTB entries with the new indices at key update to maximize BTB bandwidth utilization by the CPU pipeline. As a result, remapped entries would be considered as misses until they are reinstalled with the new indices. Since server workloads are sensitive to BTB capacity, frequent remapping would increase BTB MPKI (miss per kilo instructions), leading to significant performance overhead. CryptoBTB addresses this by preserving the preceding mappings and installing entries with the new mapping lazily. In addition, we introduce a shadow tag array, accessed concurrently with indices from the previous mapping cycle. Shadow tag array enables the reusing of the entries from the previous mapping cycle.

CryptoBTB employs a simple method to mitigate collision-based attacks. This is achieved using a 2-bit ID, uniquely identifying the owner thread and privilege level of each of the BTB entries. By ensuring entries are used only if the ID matches, we prevent collisions across privileged spaces and concurrent threads. Additionally, since the encrypted index is remapped after a context switch, collisions cannot occur across context switches as the adversary can not collide with the index.

Contribution: We make the following contribution to secure the BTB hierarchy

- We propose a low-latency index encryption method with lazy remapping to achieve a secure BTB with minimal performance overhead. The key idea is a region-based cryptographic pad design, which (a) exploits the spatial locality of BTB accesses, and (b) eliminates collisions in the encrypted indices by XORing the index with a region-based cryptographic pad. The lazy remapping enables frequent remapping with low-performance overhead.
- We provide a simple technique to prevent collision attacks by storing a thread and privilege level unique ID as part of the BTB metadata.
- We conduct a comprehensive security analysis of the CryptoBTB and examine its performance impact for several categories of single-threaded workloads. We compare our results against a non-secure exclusive hierarchical BTB and the latest state-of-the-art solution, HyBP. Our results show that CryptoBTB experiences an average performance overhead of 4.27% for these workloads, while HyBP incurs 31.89% performance overhead. Additionally, CryptoBTB increases the BTB size by 33.00%, which is 22.57% less hardware overhead than HyBP. When we consider iso-storage, CryptoBTB improves performance by 48.50% over HyBP.

The remainder of the paper is organized as follows: Section 2 provides background information on BTB and its recent exploitation techniques, highlighting the drawbacks of existing protection methods. Section 3 presents CryptoBTB's threat model, and Section 4 provides the overview of CryptoBTB. Section 5 details the design of the CryptoBTB architecture. Section 6 presents the security analysis of this architecture, Section 7 describes the methodology, and Section 8 evaluates CryptoBTB, including an analysis of various design aspects.

2 Background and Motivation

In this section, we briefly describe the working principle of the branch prediction machinery, the potential attack strategies to exploit the BTB, and the drawbacks of the proposals that mitigate these attacks.

2.1 Branch Target Buffer

The Branch Prediction Unit (BPU) in modern high-performance processors handles branch instruction identification, direction prediction, and target determination. It features a BTB for storing branch types and targets, a conditional branch direction predictor, a Return Address Stack (RAS), and an indirect branch target predictor. The BPU unit allows fetching and executing instructions speculatively from the branch target.

2.1.1 Decoupled Fetch Organization. Decoupling branch prediction from the fetch stage is important for architectures employing a scalable frontend (multi-cycle BTB and multi-cycle I-Cache) [58] and requiring instruction prefetching [59]. Instruction prefetching provides substantial speed up for high instruction footprint applications [18, 19, 34, 39, 42, 49] by reducing I-Cache miss [14]. This technique is referred to as Fetch Directed Instruction Prefetching (FDIP) [59]. Many modern processors [14, 21, 30, 31, 41] has adopted such decoupled architecture. Figure 1 presents a decoupled fetch and branch prediction where the BPU runs independently

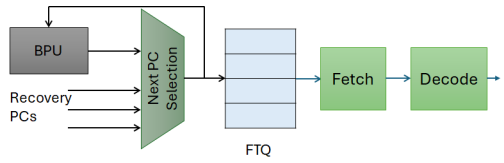


Figure 1: Fetch Target Queue to decouple BPU and Fetch.

of the fetch unit and pushes the predicted fetch addresses into the Fetch Target Queue (FTQ) [40, 60]. The backpressure in the CPU backend allows the BPU to run ahead. As a result, this organization eliminates multi-cycle BTB access latency from the pipeline, which would otherwise have been exposed due to the delay in fetching the next instruction bundle upon a BTB hit. Additionally, it enables instruction prefetching, which handles multiple misses concurrently.

2.1.2 Hierarchical BTB Organization. Large instruction footprint workloads require a significantly larger BTB to accommodate a greater number of static branch instructions. However, a larger BTB increases the access latency, affecting prefetching timeliness and exposing higher pipeline refill latency in the event of a squash. For instance, a 3-cycle 7k-entry BTB experiences a 21.16% slowdown compared to a 0-cycle BTB of the same size for workloads with diverse instruction footprints, as shown in Figure 2 (see Section 7 for details regarding these workloads). Recent CPU architectures [4, 10, 14, 21, 30, 31, 35, 41, 65] balance capacity and latency by organizing the BTB hierarchically. This hierarchy includes multiple BTBs with different sizes. We refer to the smaller BTBs as lower-level BTBs and the larger ones as higher-level BTBs, consistent with previous BTB research [54].

We elaborate on the advantage of hierarchical BTB using an experiment that configures the BTB with a smaller 1k-entry L1BTB (0-cycle latency) and a larger 6k-entry L2BTB (3-cycle latency). Figure 2 shows that the hierarchical BTB causes a 5.01% slowdown compared to the 0-cycle BTB of the same total BTB capacity, improving performance drastically compared to a long latency large single level BTB.

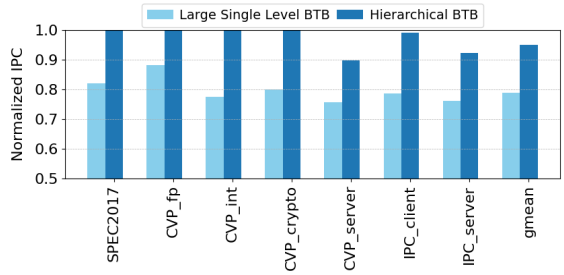


Figure 2: Performance overhead comparison between a long latency large single level BTB and a hierarchical BTB.

2.2 BTB Vulnerabilities

The BTB can be susceptible to vulnerabilities when one or more threads act adversarially in the following scenarios:

(1) Multiple applications run on the same core, either through context switching or using SMT.
(2) The user and kernel mode execution of an application run on the same core.

The attacks on the BTB result from (1) conflict within a BTB set and (2) collision due to partial tagging and the absence of PID. The attacker exploits timing side channels resulting from misfetches and type-confusions, as discussed next.

Exploiting conflict within a BTB set: This vulnerability arises from BTB eviction caused by a fill to an already full BTB set. The adversary fills a BTB set to either evict entries belonging to the victim process [12] or wait for the victim process to execute a particular branch, which subsequently evicts one of the adversary's entries [12, 13]. In the first scenario, the adversary monitors the victim's execution time; if the branch is taken, the victim's execution time increases due to the BTB misfetch. In the latter scenario, the adversary re-executes to observe if any of its branches have been evicted by the victim process. These exploits allow the adversary to learn the branch outcome, as the BTB only allocates an entry when the branch is taken. Cryptographic algorithms that use bits from the secret key as branch conditions are shown to be vulnerable to such attacks [12, 13].

Exploiting collision on a BTB entry: The use of partial tagging improves performance due to increased BTB capacity [33]. However, collisions resulting from partial tagging allow an adversarial process to misfetch and subsequently infer the training done by a victim and to maliciously train the BTB. Even with the full tag, this attack can succeed across different processes since the BTB uses virtual PC addresses. In such cases, a PID is required to avoid collisions. However, such collisions have been observed in modern processors [32, 77], proving that CPU vendors opted against adding PID due to the storage budget. Additionally, type-confusion exploitation [11] allows collision with instructions having a different type than the hit entry in the BTB, enabling stronger attacks. Recent works exploiting BTB collision vulnerability successfully demonstrate de-randomization of the KASLR [32, 71, 77], physmap KASLR [71]. Moreover, this vulnerability allows an adversary to learn about the branch direction [77], bypassing SGX enclave protection. Such attacks establish side channels either using the shared BTB or other shared components such as μop-cache, I-cache, D-cache [71, 77]. Allowing instructions from the mispeculated branch target to flow through the early pipeline stages before re-steering enables the latter side channels, even when a *non-branch* instruction collides in the BTB [71, 77]. In addition, attacks shown in [73] successfully exploit type-confusion with *non-branch* instructions to learn branch outcomes of a victim application. This attack exploits a property of the BTB, where a non-branch instruction deallocates the BTB entry it collides with.

2.3 Pitfalls of the available solutions

Several prior works [48, 76, 78, 79] have proposed content and index randomization to secure the BTB against these attacks. Conflict-based attacks require encrypting the BTB index to randomize the address-to-set mapping. To eliminate the encryption latency, [48, 78] compute XOR between the index and a secret key. However, these schemes require frequent remapping, as they cannot disperse

entries belonging to the same set originally across different sets after randomization [48, 57]. On the other hand, [76] presents a new low-latency cipher designed to address the limitations of static encryption. However, secure systems should rely on lightweight ciphers, such as PRINCE, QARMA, which have undergone rigorous security analysis, with no efficient attacks reported [36]. In contrast, [76] proposes a custom lightweight cipher that has not undergone such analysis. This presents a significant security risk. For example, the low-latency cipher LLBC, proposed in [56] for efficient randomization of address-to-set mapping in the last level cache, was later discovered to be vulnerable to cryptanalytic attacks [24].

HyBP stands out by using a strong encryption method to encrypt the index and extending the remapping interval. This hybrid solution, shown in Fig 3(a), is designed to secure a hierarchical BTB by employing both duplication and randomization. It randomizes the BTB set mapping and its contents. To minimize the frequency of remapping the lower-level BTB, it duplicates the lower-level BTB in addition to applying randomization. By incorporating both duplication and randomization, the solution reduces and randomizes the information flow to the higher-level BTB, thereby extending the remapping interval.

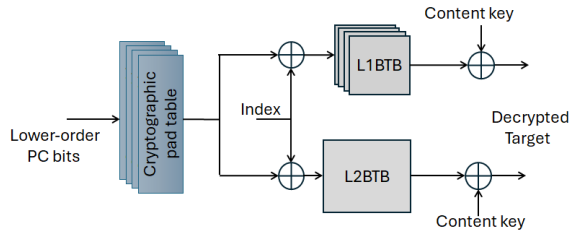


Figure 3: HyBP design overview

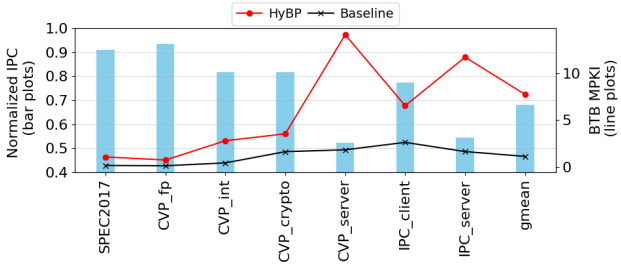


Figure 4: Performance overhead of HyBP

To facilitate a low-latency index encryption scheme, HyBP precomputes cryptographic pads for the least significant bits of the PC using a strong low-latency block cipher and a secret key. It stores them in a thread and privilege-private cryptographic pad table, referred to as the "code book" in the original work. The number of entries in this table equals the number of sets in the largest BTB. Each thread uses the original low-order PC bits to access the table, retrieve the pad, and compute a bitwise XOR with the plaintext (original) index to determine the encrypted index. Additionally, the tags and targets are encrypted using a content key. During a context switch, the cryptographic pad table is flushed and populated using a new key, and the content key is also updated, preventing both conflict and contention attacks across the context switch. However, this encryption scheme has three major drawbacks, which increase the BTB MPKI by 6.64 on average and result in an overhead of 31.89%, as shown in Figure 4. These drawbacks are as follows

- **Collision among distinct plaintext indices:** We have observed large number of collisions among PCs with the same tag. Such collisions do not exist in a non-secure BTB (see Section 5.1.1). Figure 8(b) demonstrates this phenomenon utilizing a BTB with 8 sets. Such collisions result in misfetch for both branch and non-branch type instructions. Also, since non-branch type instruction deallocates the entry in case of a hit [73], the original branch instruction that inserted it misfetches subsequently.
- **Reduction in the effective BTB capacity:** Collision in mapping in encrypted indices also results in unused sets to which none of the plaintext indices maps. As demonstrated in Figure 8(b), none of the plaintext index maps to the encrypted index 4,6 and 7. This impacts server workloads (CVP_server and IPC_server) significantly, which is sensitive to the BTB capacity.
- **Non-exclusive BTB hierarchy requirement:** The many-to-one mapping of plaintext index to the encrypted BTB index also restricts the BTB hierarchy organization to non-exclusive since HyBP can not decrypt the encrypted index of an evicted entry. Therefore, evicted entries for the lower-level can not be filled into the higher-level BTB.

In addition, the current technology trend shows the size of the lower-level BTB increases across CPU generations (from 216 entries in AMD Zen2 [65] to 1.5k entries in AMD Zen4 [21]). Since HyBP proposes a 3x increase of the L1BTB for an SMT core with 2 threads, this solution will suffer from severe scalability issues as the technology progresses.

To summarize, researchers have demonstrated several vulnerabilities in the BTB, allowing attackers to learn cryptographic keys, break KASLR, and more. Existing academic proposals to mitigate these vulnerabilities are either not fully secure or experience significant slowdown. Given the importance of securing microarchitecture in recent times, we propose CryptoBTB, a low-overhead solution to secure the BTB hierarchy.

3 Threat Model

CryptoBTB considers the threats specific to conflict and collision-based attacks within the BTB, with another thread or a different privilege level being a potential adversary. We assume that the adversary shares the same core as the victim either through context switch or concurrently using SMT configuration. Furthermore, the adversary is capable of using the BTB, cache hierarchy (including both the I- and D-Cache), and *μop* cache to establish side channels.

Similar to the threat model considered for the protecting the last level cache [56], we assume that the adversary accesses the vulnerable machine remotely; therefore, a physical side channel attack is not feasible on the CPU. Hence, the adversary can not learn the plaintext index and the ciphertext index pairs. We also consider speculative execution vulnerabilities such as Spectre [44] and Meltdown [50], and other variations of these attacks [28, 43, 45, 63, 66] are out of scope for this work. In addition, we consider that

other components in the branch prediction unit, such as direction predictor, RAS, and indirect branch target predictor, are secure. Techniques proposed in the CryptoBTB can be applied to the other components of BPU, which we leave as future work to simplify the discussion for this work.

4 CryptoBTB Design Overview

CryptoBTB is a microarchitecture enhancement designed to secure the exclusive BTB hierarchy. For a hierarchical BTB architecture, without loss of generality, we assume that L1BTB is the lower-level BTB and L2BTB is the higher-level BTB. The same design can be extended if multiple lower-level BTBs are present, such as AMD zen, which has a 16-entry L0 BTB [65].

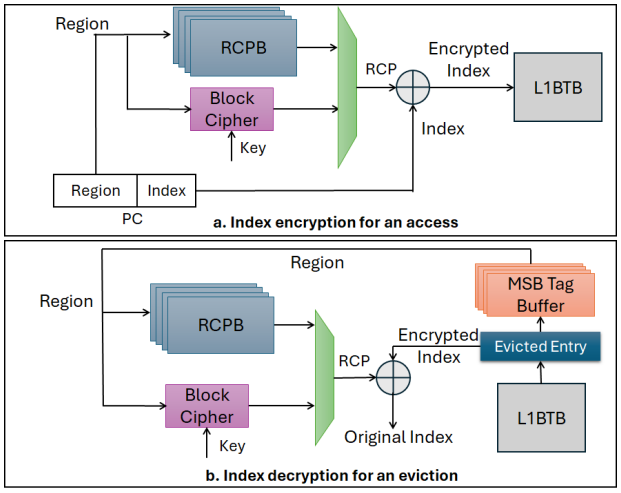


Figure 5: Overview of the CryptoBTB's index encryption and decryption.

To prevent conflict-based attacks, CryptoBTB randomizes the BTB set mapping using *cryptographic pads* to decouple the index from the encryption. Figure 5 demonstrates the index encryption for an access in the L1BTB and decryption for an eviction from the L1BTB. For this purpose, the address space is divided into regions, and each region uses a unique *cryptographic pad*, which is computed using a block cipher with a secret key and region-unique value as inputs. We identify the *cryptographic pad* as the region cryptographic pad or RCP, which is cached in a buffer (RCPB). The RCPB is physically isolated for each concurrent thread and its privilege levels. Each level of the BTB uses a distinct key; hence, each level has its own RCPB. To simplify the explanation, the design section details the techniques that CryptoBTB uses to secure L1BTB. Section 5.5 explains how the L2BTB is secured, which follows a similar design to the L1BTB with a few modifications.

Since the L1BTB is small, frequent remapping is needed to prevent an attacker from determining the eviction set. It is accomplished by updating the key. However, a key update implicitly invalidates all the L1BTB entries and causes a significant overhead when this event is frequent. CryptoBTB eliminates this overhead by preserving the previous mapping following a key update. It preserves RCPs in RCPB and the key from the previous mapping period and lazily remaps the L1BTB entries. CryptoBTB accesses L1BTB using both the current and preceding mapping. Additionally, we use *MSB Tag Buffer*, which is optimized for area and latency, to store the regions required in the index decryption step, as shown in Figure 5.

While the RCP facilitates address-to-set randomization, protecting conflict-based attacks, the collision attack is protected using thread and privilege unique ID.

5 CryptoBTB Design Details

5.1 Region Cryptographic Pad

The cryptographic pad is generated by encrypting an Initialization Vector (IV) with a secret key using a low-latency cipher as illustrated in Figure 6. The encrypted index is then computed by performing a bitwise XOR operation between the cryptographic pad and the PC plaintext index. We define a region as the address space with the full tag bits, i.e., the PC bits excluding the index and the instruction offset bits. Henceforth, we identify the full tag bits as Full-Tag. The use of the Full-Tag as IV ensures that each region has a unique cryptographic pad. As a result, the cryptographic pad is distinct for PCs mapping to the same BTB set, thereby achieving a scattering of BTB entries across different BTB sets after randomization, as shown in Figure 7.

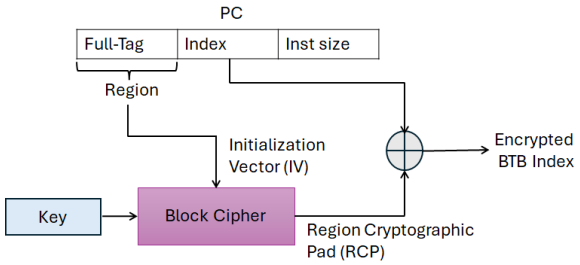


Figure 6: BTB index encryption

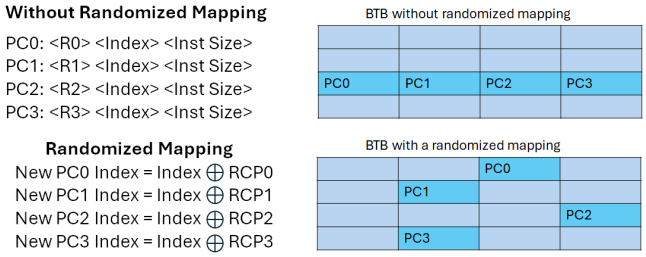


Figure 7: Address-to-set mapping before and after index encryption for PCs that originally map to the same set. RCP0, RCP1, RCP2, and RCP3 are the Region Cryptographic Pads for the region R0, R1, R2, and R3, respectively.

5.1.1 Eliminating collisions observed in HyBP. A non-secure BTB uses the plaintext index as the BTB index. Therefore, there are no collisions for different PCs within the same region (i.e., those sharing the same Full-Tag), as illustrated in Figure 8(a). However,

HyBP experiences collisions among these PCs, as shown in Figure 8(b), since the index is computed as $index \oplus encrypt(index)$. This result was obtained using the PRINCE cipher to generate the cryptographic pad for an example BTB with eight sets. CryptoBTB eliminates such collisions by computing a single pad for an entire region and using it to determine the encrypted BTB indices for the PCs within this region, i.e., index in CryptoBTB is computed as $index \oplus encrypt(region)$. Figure 8(c) demonstrates this case for a region with an RCP being set to 5.

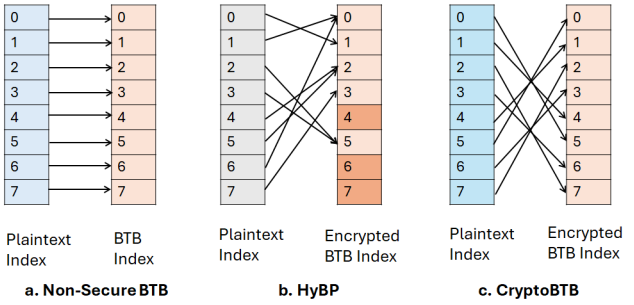


Figure 8: Examples of plaintext index to BTB index mapping for different PCs within a single region.

5.1.2 Exploiting the temporal locality of RCP. The RCP has a high temporal locality since instructions executed sequentially belong to the same region; therefore, they will use the same RCP. When an instruction crosses a region boundary or a branch diverts the execution path to a new region, the RCP of the next region is required. We leverage the temporal locality by storing the RCP in a dedicated cache, which we refer to as RCP buffer (RCPB).

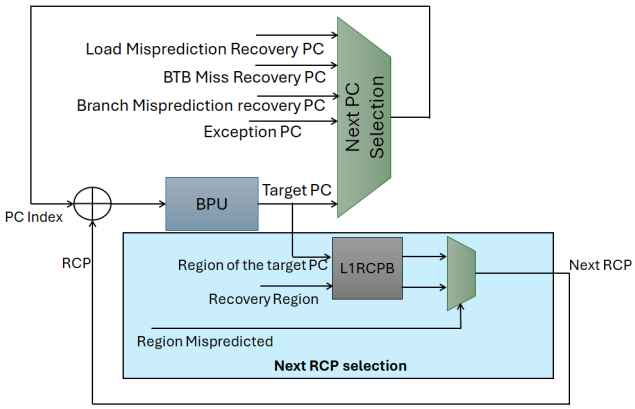


Figure 9: Circuit to compute the next RCP.

5.1.3 Reducing RCPB access latency. Since each BTB access requires accessing the RCPB, BTB access latency increases. To reduce the latency overhead further, we implement an exclusive RCPB hierarchy consisting of a small L1RCPB and a relatively large L2RCPB. The L1RCPB is accessed in parallel with the next PC selection logic. This logic selects among load misprediction recovery, exception recovery, and BPU-predicted PC, as shown in Figure 9. We adopt the next PC selection logic from AnyCore, an open-sourced universal superscalar core [1, 29]. Considering that branch and load mispredictions, along with exceptions, are rare events, we optimistically select the region of the BPU-predicted PC to access the L1RCPB. This region corresponds either to the predicted branch target if the branch is taken or to the next sequential PC in the event of a not-taken prediction. With this design choice, we partially eliminate the next PC selection from the critical path of the next RCP selection.

Misprediction/Exception recovery always results in region misprediction. The correct region (recovery region in Figure 9) accesses the RCPB hierarchy in the next cycle. In this case, CryptoBTB incurs one-cycle bubble. Note that L2RCPB is accessed after the next PC selection, incurring stalls equal to the L2RCPB access latency in case of a L1RCPB miss and L2RCPB hit. In the event that RCP misses the RCPB hierarchy, it must be computed using the block cipher, as shown in Figure 5, resulting in as many pipeline bubbles as the block cipher latency.

5.1.4 RCPB entry organization. An RCPB entry contains the following fields

- Valid: The valid bit indicates whether the entry is valid.
- RCP_{phase}: stores the *global phase*. *Global phase* is used to identify the mapping cycle (see Section 5.2 for *global phase*).
- RCP_{cur}: stores the RCP computed with the latest key.
- RCP_{prev}: stores the RCP computed with the preceding key, following a key update.

5.2 Key Change and Lazy Remapping of L1BTB

Since randomizing the BTB set mapping can only protect it until an adversary recovers the eviction set, periodic remapping is needed [56]. The remapping interval is small for the L1BTB because of its small size (see Section 7). Due to the high overhead associated with frequent key changes, we introduce lazy remapping. In this scheme, the L1BTB is searched using indices computed from both the current and preceding keys. In case of a hit with the preceding key, the entry is remapped with the latest key. Moreover, after a key update, RCPs in RCPB are also computed lazily only when a specific region is accessed. To enable lazy remapping, CryptoBTB employs a *global phase* to identify the current mapping cycle and measure how old the mapping cycles of each RCPB and the L1BTB entries are compared to it. To achieve this, we enhance the design using the following changes:

- Global phase: The thread and privilege space private *global phase* is incremented in each key update. It is reset to zero when it reaches the maximum value, determined by its bit-width.
- RCP_{phase} in RCPB: This field stores the *global phase* at the time of an RCP update. Any mismatch between the RCP_{phase} and the current *global phase* indicates the staleness of the RCP_{cur} in RCPB.
- BTE_{phase} in L1BTB: At the time of an L1BTB fill, the *global phase* is stored as part of the tag to assist in identifying an L1BTB hit (discussed in Section 5.4.3).

To facilitate searching the L1BTB with the previous key, CryptoBTB is enhanced with the following components:

- RCP_{prev} in RCPB: This field stores the RCP computed with the preceding key (Key_{prev}).
- Key_{prev} buffer: This buffer stores the preceding key after a key update.
- Shadow L1BTB Tag Array: The shadow L1BTB tag array mirrors the L1BTB tag array and is accessed simultaneously with it. However, the shadow L1BTB tag array uses the index computed using RCP_{prev}. If there is a hit, the L1BTB data array is accessed in the next cycle using this encrypted index to retrieve the target, resulting in a one-cycle pipeline bubble.

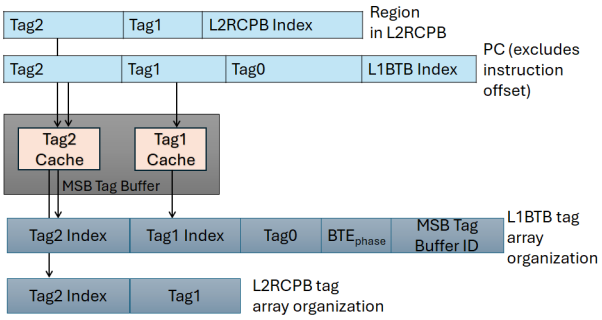


Figure 10: Tag array organization of L1BTB and L2RCPB

5.3 MSB Tag Buffer

CryptoBTB includes *MSB Tag Buffers* to reduce the tag overhead in the L1BTB and L2RCPB. This structure is physically isolated among concurrent threads and privilege spaces. Using server traces, we observe that the higher-order PC bits have low variation. For instance, after partitioning the Full-Tag for a BTB with 1024 sets into three components: Tag0 (9 bits), Tag1 (8 bits), and Tag2 (the remaining higher-order bits), we observe that the maximum number of unique partial tags are 509, 19, and 4, respectively among workloads within IPC_server. Given the minimal variation in Tag1 and Tag2, CryptoBTB employs thread and privilege-private fully associative tag caches to store them, as illustrated in Figure 10. Indices to these tag caches are stored as part of the L1BTB tag, along with other metadata. It assigns thread and privilege space unique 2-bit ID to identify *MSB Tag Buffer*. We coin this ID as the *MSB Tag Buffer ID*.

This organization is essential to minimize collision within the same process and privilege space during access into the L1BTB hierarchy and eviction from the L1BTB into the L2BTB. Moreover, *MSB Tag Buffer ID* eliminates collision across concurrent threads and privilege spaces. During an L1BTB update, the *MSB Tag Buffer* provides the indices for Tag1 and Tag2, and the L1BTB tag is created by concatenating these indices with Tag0, BTE_{phase} and *MSB Tag Buffer ID* as shown in Figure 10. In case of a miss in *MSB Tag Buffer*, at the time of BTB update, CryptoBTB allocates new entries in the *MSB Tag Buffer*. At the time of access, CryptoBTB compares the concatenated tag with the tag stored in the L1BTB.

Moreover, we utilize this buffer partially to reduce the tag overhead of the L2RCPB by partitioning its Full-Tag into two. We partition the Full-Tag in such a way that Tag2 aligns for both the L1BTB and L2RCPB as shown in Figure 10. The alignment in the partial tag bits enables CryptoBTB to implement single *MSB Tag Buffer* for each thread and privilege space. The *MSB Tag Buffer* is accessed in parallel with access to the L1BTB, L2RCPB. L1RCPB does not use this optimization since its access latency is small.

5.4 Working Principle of L1BTB

In this subsection, we describe the working principle of L1BTB. These steps are illustrated in Figure 11.

5.4.1 ***Key update.*** To facilitate the update of the L1BTB key, we implement an L1BTB access counter that increments with each L1BTB access. The key update process follows these steps: (1) upon an L1BTB access, the access counter value is checked to verify if it has reached the predetermined L1BTB access threshold. (2) If the threshold is reached, the current key (Key_{cur}) is copied to the previous key (Key_{prev}). (3) A new random value generated by an on-chip key generation module is assigned to Key_{cur}, (4) the *global phase* is updated, and (5) the L1BTB access counter is reset. Figure 11(a) demonstrates these steps.

5.4.2 ***RCPB update.*** The L1RCPB is updated in the event of a miss or if it's a hit, but the RCP_{phase} is stale, i.e., the RCP_{phase} does not match *global phase*. In such cases (1) both the RCP_{cur} and RCP_{prev} are computed for the corresponding region using Key_{cur} and Key_{prev}, respectively. Finally, (2) the RCP_{cur}, RCP_{prev}, and the *global phase* as RCP_{phase} are stored in the RCPB. L2RCPB is updated upon an eviction from L1RCPB. During an L2RCPB write, in case of a miss in the Tag2 cache in the *MSB Tag Buffer*, this is also updated. Figure 11(b) illustrates the steps required for an RCPB update.

5.4.3 ***L1BTB access.*** Accessing the L1BTB involves computing the current and preceding encrypted indices using the RCP_{cur} and RCP_{prev}, respectively. Figure 11(c) demonstrates the steps for an L1BTB access. The following steps are used: (1) the L1BTB tag and shadow tag arrays are searched using the current and preceding encrypted indices, respectively. (2) In the case of a hit in the shadow L1BTB tag array, the L1BTB data array is accessed in the next cycle to retrieve the target. Along with the thread and privilege unique ID, Tag2 and Tag1 cache indices from the *MSB Tag Buffer* and Tag0, CryptoBTB uses RCP_{phase} and $RCP_{phase} - 1$ to construct the tags, which searches the L1BTB and shadow L1BTB tag array, respectively. Therefore, for a hit in the tag array, BTE_{phase} must be equal to RCP_{phase} and for a hit in the shadow L1BTB tag array BTE_{phase} must be equal to $RCP_{phase} - 1$. Note that $RCP_{phase} - 1$ specifies the preceding phase. The additional phase check ensures that the entries were inserted during the corresponding *global phase*.

CryptoBTB can not access entries with BTE_{phase} lower than $RCP_{phase} - 1$. As a result, access to such entries will be considered misses. This does not affect correctness since the pipeline will be re-steered in the subsequent stages upon identifying BTB misfetch.

5.4.4 ***L1BTB update.*** The L1BTB is updated when a taken branch is missing, or there is a hit in the shadow L1BTB tag array. The update process follows these steps : (1) both the tag array and shadow L1BTB tag array are accessed for a possible hit. (2) If there is a hit in the tag array, an update is not required, but in the event of a hit in the shadow L1BTB tag array, the L1BTB data array is accessed in the next cycle to retrieve the target before invalidating

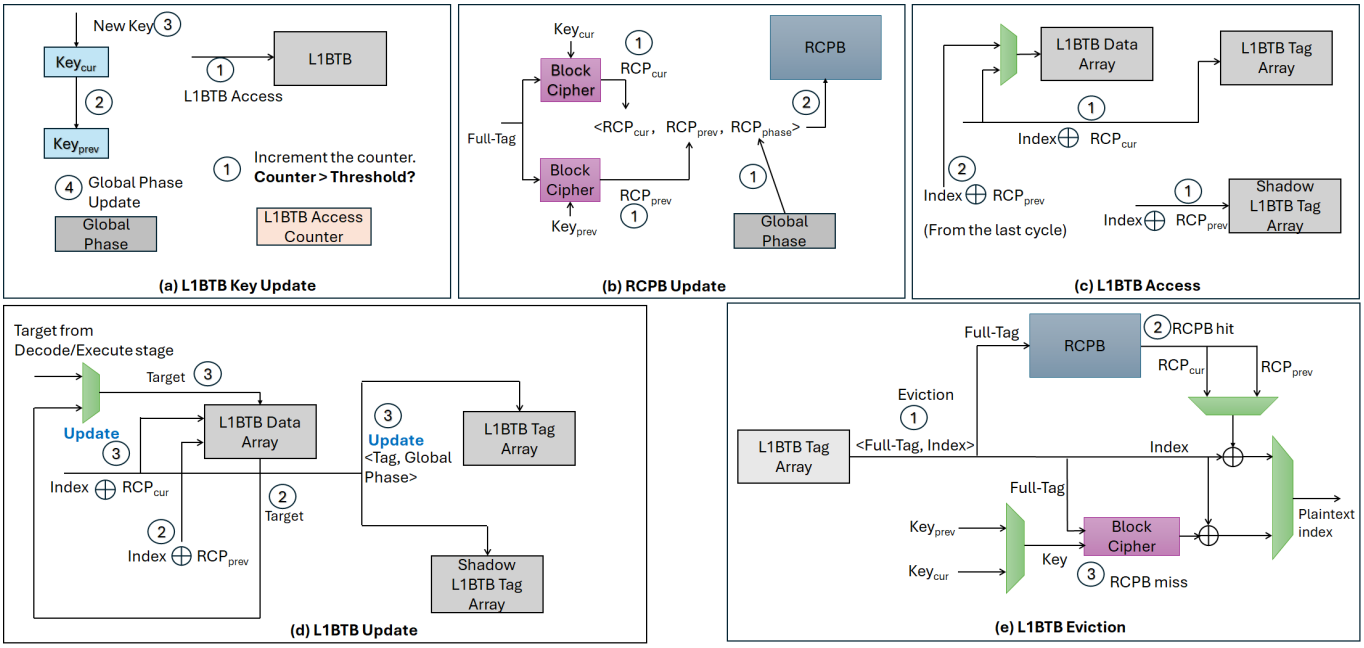


Figure 11: Working principle of the L1BTB

the stale entry. (3) Subsequently, the L1BTB is updated with the target for the current encrypted index. If there is a miss also in the shadow L1BTB tag array, the correct target is updated from the later pipeline stages. The index is computed using the RCP_{cur} during the update. The L1BTB tag arrays store the *global phase* in the BTE_{phase} field, the thread and privilege space unique ID, Tag2 and Tag1 cache indices, Tag0, at the time of the L1BTB update. CryptoBTB updates the *MSB Tag Buffer* in case of a miss before retrieving the Tag2 and Tag1 cache indices. Figure 11(d) presents these steps.

CryptoBTB flushes the L1BTB, along with the shadow L1BTB tag array, when *global phase* reaches maximum to ensure correct identification of stale phases after a wraparound of the *global phase*.

5.4.5 ***L1BTB eviction.*** In the event of an L1BTB eviction, CryptoBTB decrypts the encrypted index of the evicted entry by computing XOR between its L1BTB index and RCP of the corresponding region, and inserts it into the L2BTB. The decryption happens using the following steps: (1) the index and tag of the evicted entry are retrieved, and CryptoBTB uses Tag1 and Tag2 cache indices stored in the L1BTB tag to retrieve Tag1 and Tag2 from the *MSB Tag Buffer*. The region of the evicted entry is determined by concatenating Tag2 and Tag1 with Tag0. (2) RCPB is accessed to retrieve RCP_{cur} and RCP_{prev} for the corresponding region, and the correct RCP is selected according to the BTE_{phase} and RCP_{phase} to perform a bit-wise XOR with the encrypted index to compute the plaintext index. If BTE_{phase} is equal to the RCP_{phase}, RCP_{cur} is chosen, but if it is equal to RCP_{phase}-1, RCP_{prev} is chosen for decryption. (3) In case of a RCPB miss, the RCP is computed before determining the plaintext index. In this case, successful decryption requires the BTE_{phase} to be either the *global phase* or *global phase*-1 and utilizes Key_{cur} and Key_{prev} respectively. If BTE_{phase} is older than the preceding phase, CryptoBTB does not insert the evicted entry into the L2BTB. We demonstrate these sequences of steps in Figure 11 (e).

5.5 Securing L2BTB

We randomize the L2BTB address-to-set mapping similar to the L1BTB, and enhance it with an RCPB hierarchy. However, the L2BTB does not need a shadow tag array since the remapping interval for L2BTB is extended as L1BTB filters most of the L2BTB traffic (see Section 6 for discussion on the remapping interval). Furthermore, we partition the Full-Tag required for the L2BTB similarly to the L1BTB, as shown in Figure 10. The Tag2 and Tag1 of L2BTB and L1BTB align so that they can share the same *MSB Tag Buffer*. Similar to the L1BTB, the L2BTB also stores *MSB Tag Buffer* ID as part of the tag.

5.6 Action at context switch

To prevent attacks across a context-switch, at the time of the context switch, CryptoBTB updates the key and invalidates the RCPB hierarchy to implicitly invalidate the BTB hierarchy, invalidates *MSB Tag Buffers* belonging to the user and kernel spaces for the last thread to prevent an adversary from learning the pages that the last victim executed, and invalidates L1BTB to identify stale entries correctly. Furthermore, it also resets *global phase*.

6 Security Analysis of CryptoBTB

This section examines the security aspects of CryptoBTB.

6.1 Analysis of conflict attack protection

6.1.1 L1BTB. CryptoBTB employs a random replacement policy, making it difficult to identify the eviction set. We show the pseudo-code for the eviction set determination algorithm in Algorithm 1.

The remapping interval equals the total number of accesses required to find #ways distinct PCs for the eviction set. Similar to a cache, the expected number of accesses needed to find the eviction set is equal to $2 * W_{L1BTB} * N_{L1BTB}$ as demonstrated in [57] [1], where W represents the number of ways, and N denotes the total size. For the configuration (refer to Section 7) considered for the evaluation, the remapping interval is 18342 BTB accesses.

Algorithm 1 Algorithm to find a eviction set

```
procedure FindEvictionSet(PC_v)
    E = {}
    for i=0 to #ways − 1 do                          ▷ Outer-loop
        while ExecTime(PC_v) < Threshold do          ▷ Inner loop
            Execute PC_v
            Execute randomly chosen PC: PC_x
        end while
        insert(E,PC_x)
    end for
    return E
end procedure
```

6.1.2 L2BTB. Given the exclusivity of BTB hierarchies, the L1BTB filters most of the traffic to the L2BTB. Since L2BTB is updated with a new entry upon an L1BTB eviction, constructing a contention-based attack on the L2BTB requires evicting relevant entries from the L1BTB. Hence, both the determination of the eviction set and the attack itself require the adversary to evict entries from the L1BTB and insert them into the L2BTB. An adversary requires $2 * W_{L2BTB} * N_{L2BTB} * N_{L1BTB}$ expected number accesses to the BTB hierarchy to determine an eviction set in the L2BTB. It uses a similar formula to the one that we use for the L1BTB, except for the extra multiplication of N_{L1BTB} to account for the expected number of accesses required for evicting an L1BTB entry with a random replacement policy. The remapping interval for the L2BTB is approximately 150 million for the configuration used for our evaluation.

6.2 Analysis of collision attack protection

CryptoBTB prevents collision-based attacks using the thread and privilege unique 2-bit ID. This ID check ensures that collisions never happen between concurrent SMT threads and privilege levels. On the other hand, collisions between two different processes across the context switch are prevented by the index remapping at the time of the context switch and RCPB flush, implicitly invalidating L1BTB and L2BTB.

6.3 Protection across context switch

To prevent cross-context attacks, CryptoBTB updates keys for both L1BTB and L2BTB and invalidates the RCPB hierarchy during context switches. This prevents reusing older entries by an adversary process after a victim application is switched out. Even if an adversary discovers colliding entries from the victim's last context, this information will be useless, as the victim context would use a new mapping when it is switched back into the core. Additionally, flushing the *MSB Tag Buffer* mitigates vulnerabilities that aim to leak the instruction page executed by the victim.

[1]For brevity, we skip the detail evaluation of the formula. We request the readers to refer to [57] for the evaluation.

6.4 Analysis of other components in CryptoBTB

6.4.1 ***Hits in the shadow tag array.*** To analyze the shadow tag array, we consider a scenario where the attacker has identified all W distinct PCs for the eviction set and executes the final terminating condition of Algorithm 1's inner loop to measure the access time for PC_v, the target PC for eviction set discovery.

At this stage, the algorithm randomly evicts one PC from the eviction set, as CryptoBTB employs a random replacement policy. The key is then updated as illustrated in Figure 12. After the key update, the set contains W-1 PCs from the eviction set, which would utilize the shadow L1BTB tag array for a hit. However, for a future conflict-based attack, the attacker must populate the set with the entire eviction set, requiring the attacker to execute all PCs from the eviction set, as they cannot determine which PC was last evicted by PC_v due to the random replacement policy. Once all eviction set PCs are executed, they are re-mapped with the current key, as CryptoBTB immediately updates the mapping if an entry is found with the previous mapping. Consequently, the shadow L1BTB tag array's use remains secure.

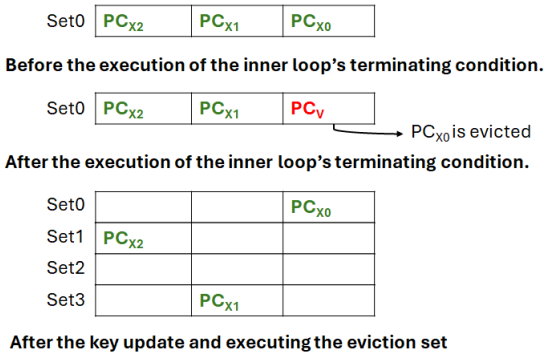


Figure 12: L1BTB states while executing the eviction set discovery algorithm, followed by a key update where the eviction set contains PC_{x0}, PC_{x1} and PC_{x2}.

6.4.2 ***Speculative access to the L1RCPB.*** CryptoBTB reduces RCPB hierarchy access penalties by speculatively using the BPU target. This incurs a one-cycle latency when a recovery PC is chosen. While this solution increases recovery latency, it doesn't introduce new vulnerabilities, as adversaries can already measure recovery latency due to pipeline flush and re-steer [73, 77].

6.4.3 ***Speculative BTB update vulnerability.*** Prior works demonstrated potential vulnerability due to the speculative update to the BPU state [69, 75]. To mitigate such vulnerability, prior mitigations [47, 75] proposed updating the BPU state at commit. Similarly, speculatively updating BTB following a mispeculated branch is a potential source of vulnerability. One possible solution to prevent speculative update vulnerability of the BTB state is to update the BTB at commit, adopting existing solutions. However, in this work, we assume that speculative execution vulnerability, which relies

on the mistraining of the conditional branch predictor, is not part of the threat model. Therefore, CryptoBTB does not impose any particular restriction on the BTB update.

The components of CryptoBTB, such as RCPB hierarchy and *MSB Tag Buffer*, are protected against concurrent threads and other privilege levels as they are isolated among them.

7 Methodology

Table 1: Simulation configurations

BTB sizes	L1BTB: 1.5k entries. 256 sets, 6 ways L2BTB: 7k entries. 1024 sets, 7 ways L1BTB Tag : (([43 : 28] ⊕ [27 : 12]) « 2) | [11 : 10] L2BTB Tag : [43 : 28] ⊕ [27 : 12] RAS: 64 entries Indirect Branch Predictor: 4096 entries
Direction Predictor	TAGE-SC-L
Fetch Target Queue	512 entries One entry per instruction
Frontend	8-Wide Fetch; 4-Wide Decode; 6-Wide Dispatch
Backend	ROB/LQ/SQ/IQ entries = 320/88/64/120 8-Wide Execute; 8-Wide Retire
Memory	L1 Inst Cache:32kB, (64 sets/8 ways), 3cycle latency L1 Data Cache:32kB, (64 sets/8 ways), 3cycle latency L2 Cache:1MB, (1024 sets/16 ways), 14cycle latency L3 Cache:4MB, (4096 sets/16 ways), 50cycle latency 32GB DRAM: 3200 MHz, quad-channel ITLB: 32 sets, 4 ways DTLB: 32 sets, 4 ways STLB: 128 sets, 12 ways
CryptoBTB	L1RCPB (L1BTB/L2BTB): 16 sets/Direct Mapped L2RCPB (L1BTB/L2BTB): 64 sets/8 ways Block Cipher: PRINCE [27], 3 cycle latency *Tag1 Cache*: 32 sets, FA; *Tag2 Cache*: 8 sets, FA Global Phase bit-width: 8

We evaluated CryptoBTB using ChampSim [3], a trace-based simulator (commit *2b8d3fc*). Several recent academic works [16, 37, 51, 54] used this simulator to study BTB organization in servers for workloads with high instruction footprints. It offers an excellent infrastructure for analyzing such workloads due to the FDIP support and publicly available server traces. In recent years, researchers have also used this simulator to study prefetchers and memory hierarchies [20, 22, 52, 53]. ChampSim has a few limitations compared to cycle-level simulators like gem5 [23]. For example, ChampSim does not simulate the wrong path. However, we chose ChampSim over gem5 for a set of reasons: (1) gem5 does not support decoupled fetch, and (2) the availability of a large number of server traces from the industry that are compatible with ChampSim.

Champsim re-steers at decode in case of either misfetch or type-confusion of direct jump, direct call, and conditional branch. We enhance the simulator with recovery at the decode stage for the indirect call, indirect branch, and return instructions upon type-confusion misprediction by the BTB. Furthermore, we update the model to deallocate a BTB entry if it conflicts with a non-branch instruction, which aligns with recent CPU architectures [73].

ChampSim simulator configuration for the evaluation closely follows recent high-performance processors as shown in Table 1. To ensure that the L1RCPB access latency is less than or equal to that of the next PC selection circuit (see Section 5.1.3), we size the L1RCPB appropriately. We synthesized the PC selection circuit using Synopsys DC Compiler and calculated the L1RCPB access latency with OpenRAM [38] using the same technology node. Our simulation indicates that an L1RCPB with 16 sets is the largest buffer configuration that meets the access latency requirement. For both baseline and HyBP, we employ partial tags in L1BTB and L2BTB as described in Table 1. The tag length was optimized to allow eviction into the higher-level and to minimize collisions in our studied workloads, with further increases achieving negligible improvements.

We evaluate CryptoBTB using traces from SPEC2017, the Championship Value Prediction (CVP) [2], and the Instruction Prefetcher Championship (IPC) [7]. Note that the workloads in IPC are a subset of those in CVP, which we exclude from the CVP analysis. We utilize publicly available SPEC2017 simpoint traces [8] compatible with ChampSim, used in recent research [22]. We simulate 476 traces in total, comprising 55 floating-point, 56 integer, 28 crypto, and 200 server workload traces from CVP. Additionally, it includes 8 client and 35 server workload traces from IPC, along with 94 SPEC2017 workload traces. For each simulation, we use 50 million instructions for warmup and simulate the next 50 million instructions in detail.

HyBP is configured similarly to the baseline. In HyBP, L1BTB and L2BTB utilize a shared cryptographic pad, as illustrated in Figure 3. The number of entries in the cryptographic pad table is equal to the number of L2BTB sets. For L2BTB, HyBP uses plaintext index bits to access the cryptographic pad table. However, the L1BTB utilizes a portion of the lower-order Full-Tag bits along with the plaintext index to access the cryptographic pad table, as the table contains more sets than the L1BTB (i.e., the index bit-width of L1BTB is smaller than the index bit-width of the cryptographic pad table).

8 Evaluation

In this section, we provide a performance analysis of CryptoBTB and compare it with the HyBP. We also explore various design spaces to understand CryptoBTB's performance overhead. For simplicity in the discussion, we group CVP_server and IPC_server into "server workloads" as both contain traces from server-class workloads and group and classify the remaining workloads as "low-to-moderate instruction footprint workloads".

Figure 13 illustrates the normalized IPC for both the CryptoBTB and HyBP architectures in relation to the baseline with an exclusive BTB hierarchy. The chart also displays the BTB MPKI for these workloads. The BTB MPKI considers BTB misses that miss both L1BTB and L2BTB. Figure 13 demonstrates that CryptoBTB significantly outperforms HyBP across all workload categories. On average, CryptoBTB incurs an overhead of 4.27%, while HyBP suffers from a 31.89% overhead. Moreover, HyBP increases BTB MPKI significantly in relation to the baseline non-secure BTB, while CryptoBTB experiences a minor increment.

In addition to the increase in BTB MPKI, non-branch instruction collision in BTB also impacts the performance of HyBP. The average 12.32% overhead for the workloads with low-to-moderate instruction footprints is primarily caused by higher collisions in HyBP. Server workloads experience an average overhead of 47.44% due to significant BTB MPKI degradation caused by collisions, underutilization, and non-exclusivity in the BTB hierarchy. Combined with the substantial degradation in BTB MPKI and the impact of BTB misses on FDIP, HyBP experiences massive slowdowns for these workloads. In contrast, CryptoBTB solves all these drawbacks observed in HyBP and achieves nearly identical performance for the low-to-moderate instruction footprint workloads (1.32% overhead on average). The overhead for the server workloads is a little bit

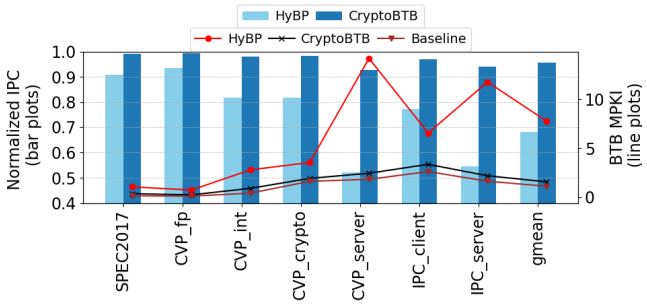


Figure 13: Performance overhead of HyBP and CryptoBTB in relation to the baseline.

higher (7.2% on average), but still, it is 40.24% lower than HyBP. Unlike HyBP, CryptoBTB does not suffer from collisions; therefore, the performance degradation is only due to increased BTB misses. Server workloads rely on the BTB to prefetch instructions into the I-Cache using FDIP (see Section 2.1.1). For example, IPC1_server has an I-Cache MPKI of 71.237, while SPEC2017 workloads see only 0.0019. FDIP reduces fetch stalls by enabling instruction prefetching, thereby minimizing performance overhead caused by ICache misses. As a result, extra BTB misses by CryptoBTB introduce additional fetch stalls. This explains the slight performance overhead observed in server workloads with CryptoBTB, compared to workloads with low-to-medium instruction footprints.

8.1 Context Switch

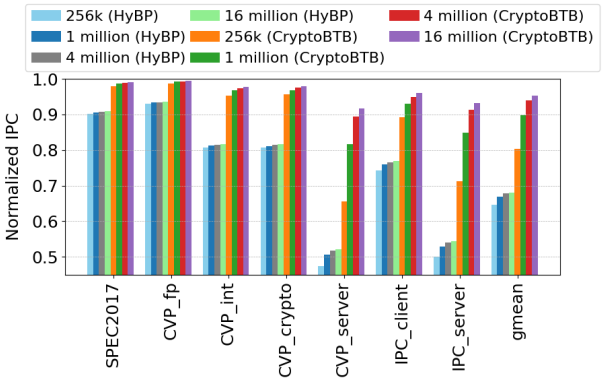


Figure 14: Performance overhead of HyBP and CryptoBTB for different context switching intervals.

CryptoBTB updates the key upon a context switch and flushes the L1BTB and RCPB hierarchies, which implicitly invalidates entries in the L2BTB. Moreover, it also flushes *MSB Tag Buffer*. To understand the impact of the context switch, we vary the context switch interval and illustrate the results for both HyBP and CryptoBTB in Figure 14. In this study, we choose context switch intervals of 256k, 1 million, 4 million, and 16 million instructions. Note that the RAS, indirect branch target predictor, and direction predictor are not flushed at the context switch, as our threat model considers these components to be secure.

The performance overhead for HyBP due to the context switch is worse than the overhead shown in Figure 13, as expected. For CryptoBTB, the context switching impacts server workloads the most due to its impact on FDIP. In contrast, the average performance overhead of 3.06% for the rest of the workloads, even at a context switch interval of 256k instructions, is insignificant. As we increase the context switch interval, performance overhead decreases. The average performance overhead for the server workloads is 8.17% for a context switch interval of 16 million instructions, which constitutes a 4ms timeslice assuming a 4GHz processor and IPC of 1. This is 0.97% higher than the overhead CryptoBTB experiences without any context switch.

8.2 Sensitivity Study

To understand the impact of various design configurations of the CryptoBTB, we vary various parameters of the CryptoBTB.

8.2.1 Sensitivity to RCPB Size. The RCPB hierarchy incurs a 1-cycle penalty for an L1RCPB miss followed by an L2RCPB hit. As a result, CryptoBTB is not very sensitive to the L1RCPB miss since a large L2RCPB supports it. If there is also an L2RCPB miss, the penalty increases to 3 cycles, which is the RCP computation latency. Since an RCPB miss causes a 3-cycle pipeline stall, an increase in RCPB MPKI has less impact on server workloads compared to a BTB miss, which can lead to a fetch stall. Note that since CryptoBTB maintains RCPB hierarchy for both L1BTB and L2BTB, it has a 1-cycle penalty if there is a miss in either of the two L1RCPBs and a penalty of 3 cycles in case of a miss in either of two L2RCPBs. To simplify design space exploration, we keep the L1RCPB size the same for both L1BTB and L2BTB. Similarly, the L2RCPB size is kept consistent as well.

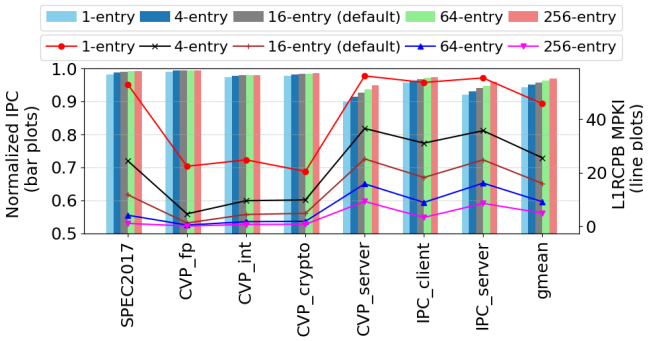


Figure 15: Performance sensitivity to L1RCPB size.

Figure 15 shows the performance overhead and the MPKI for L1RCPB of L1BTB when varying L1RCPB size. This figure shows that even a small L1RCPB does not degrade performance significantly due to the presence of a large L2RCPB. CryptoBTB with 1-entry L1RCPB results in a 5.81% overhead, which is only 1.54% higher than the observed overhead by the default CryptoBTB configuration.

Figure 16 presents the performance overhead associated with varying L2RCPB size. Similar to the L1RCPB, the performance overhead does not change drastically with the changes in the L2RCPB size. Even a small L2RCPB of 256 entries is able to accommodate

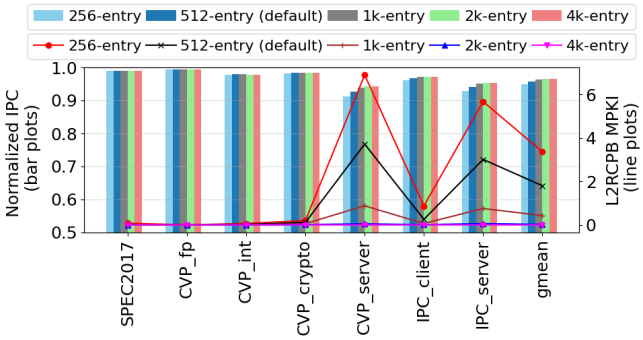


Figure 16: Performance sensitivity to L2RCPB size.

all the entries for the low-to-moderate instruction footprint workloads, as L2RCPB MPKI for these workloads is close to zero. For server workloads, the MPKI drops to zero for 2k-entry and 4k-entry L2RCPB.

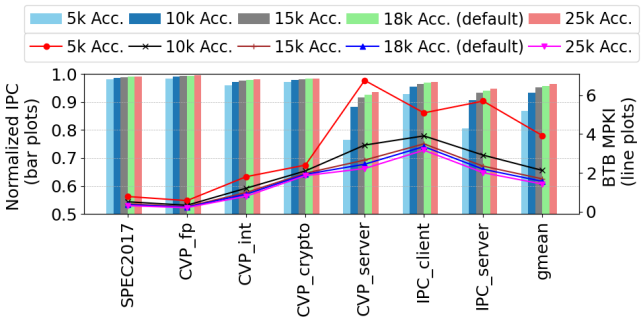


Figure 17: Performance sensitivity to the remapping interval.

8.2.2 Sensitivity to L1BTB Remapping Period. To understand CryptoBTB's potential performance impact to prevent future attacks that require more frequent remapping, we vary the remapping interval of the L1BTB, which is shown in Figure 17. The graph also includes the BTB MPKI for each configuration to illustrate the increment in BTB MPKI with shorter remapping intervals. We observed that the average BTB MPKI is higher when the remapping interval is 5000 BTB accesses, but it reduces drastically when the remapping interval is increased to 10000. Even with a remapping interval of 5000, CryptoBTB has 18.62% lower performance overhead compared to HyBP.

8.2.3 Sensitivity to MSB Tag Buffer Size. To understand the impact of the *MSB Tag Buffer*, we varied the size of the *Tag1 Cache*. A smaller *Tag1 Cache* may not cover the entire footprint, leading to higher miss rates in the L1BTB and incorrect decryption of the evicted L1BTB indices. Both factors significantly impact performance, as shown in Figure 18. CryptoBTB with a 4-entry *Tag1 Cache* incurs 37.69% overhead on average. The server workloads contribute more to the average performance degradation. However, increasing the size of this cache improves performance, with a 32-entry *Tag1 Cache* being sufficient to store all unique Tag1 partial tags, leading to performance saturation.

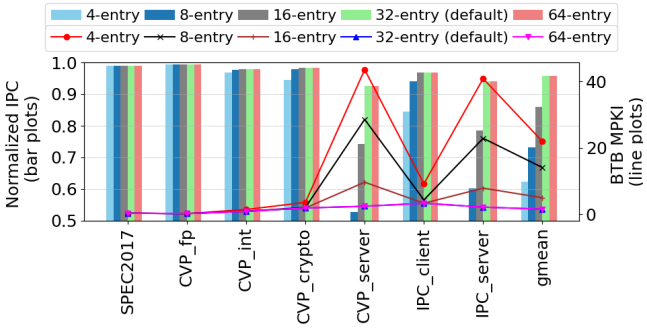


Figure 18: Performance sensitivity to the Tag1 Cache size.

We additionally evaluated CryptoBTB's performance sensitivity in relation to the bit width of *global phase* and block cipher latency. The average performance overhead for the phase bit-width of 5 and 9 are 4.88% and 4.23%, respectively. Our experiments show only a 1% degradation in performance between cipher latencies of 1 and 5 cycles. These evaluations indicate that CryptoBTB's performance does not change drastically when the configurations of these two components are varied.

8.3 Hardware Overhead

Table 2 shows the breakdown of the hardware overhead for this design, considering an SMT configuration with 2 threads. The major contributors are the L1BTB tag array and the RCPBs. For a 5-level page table, the size of the baseline is computed considering a 57-bit target, with bits per entry using a similar structure as shown in [64]. The percentage overhead for CryptoBTB is 33.00%, while HyBP's area overhead is 55.57%. L1BTB tag arrays and L2RCPB are the primary contributors to CryptoBTB's area overhead. This overhead can be reduced by varying several parameters with trading off performance. For instance, reducing the number of sets for the L2RCPBs to 32 lowers the overhead to 23.39%, with an additional performance overhead of 1.7%. Since CryptoBTB only duplicates the L1BTB tag array, it is scalable, as opposed to HyBP, which proposes a 3x duplication of the L1BTB.

Table 2: Hardware Overhead Breakdown

L1BTB TAG	7.875 kB
$L1RCPB_{L1BTB}$	0.52 kB
$L1RCPB_{L2BTB}$	0.38 kB
$L2RCPB_{L1BTB}$	10.25 kB
$L2RCPB_{L2BTB}$	5.75 kB
MSB Tag Buffer	0.0586 kB
L2BTB	2.625 kB
Total	27.46 kB

9 Related Work

In this section we discuss existing works for protecting speculative execution attacks and index randomization proposals for the caches.

9.1 Speculative Execution Attack Mitigations

A speculative execution attack exploits speculative fetch and execution of instructions from the wrong execution path to create a side

channel in various shared structures such as cache hierarchy, including D-Cache and I-cache. Spectre [44] and Meltdown [50] are the most studied attack variants exploiting speculative execution. Recently, researchers have proposed various architectural solutions to protect the cache hierarchy against these attacks. Ghostminion[15] Invisispec [72], Cleanupspec [62] enhance the cache hierarchy to be re-populated upon branch resolution to prevent side-effect of speculatively accessed data. STT[75] and NDA [69] prohibit execution and forwarding of unsafe speculative instructions (speculative instructions with side-effects in the shared structures) until they become bound to retire. The overhead caused by stalling unsafe load instructions is partially mitigated by predicting their location within the cache hierarchy [74] and address [47], using predictors trained with information from committed instructions.

9.2 Randomized Cache Architecture

Randomizing the cache specifically to protect against attacks has been widely studied in recent times. CEASER [56] and CEASER-S [57] propose randomizing the cache using a low latency cipher, LLBC. Unfortunately, this cipher was later found to be linear and vulnerable to cryptanalytic attacks [24, 55]. Scatter-cache [70] used skewed associativity in addition to randomizing the cache. However, [55] presented a method to create probabilistic eviction sets, disproving the claim of a long remap period for skewed cache. MIRAGE [61], on the other hand, proposed a skewed randomized cache with global eviction to prevent set associative evictions. All of these solutions increase the access latency with the encryption latency, making them unsuitable for BTB. Additionally, MIRAGE incurs additional latency on top of the encryption latency due to indirection. L1cache randomization schemes, RPCache [67], NewCache [68] require marking the page table as protected, which requires reliance on the OS and may not be desirable for secure enclave execution.

10 Conclusion

Current solutions for securing BTB are either not fully secure or experience significant performance overhead. In this work, we propose CryptoBTB, a secure hierarchical BTB that is suitable for workloads with diverse instruction footprints. CryptoBTB separates the index encryption from the index using cryptographic pads. CryptoBTB exploits the temporal locality of the pads to cache recently used pads to reduce encryption latency. Moreover, CryptoBTB's index encryption technique eliminates all the drawbacks of HyBP, a state-of-the-art solution to protect the BTB. This design secures the BTB with an average performance overhead of 4.27% while increasing the BTB size by 33.00%. Compared to HyBP, which has a substantial 31.89% performance overhead and a 55.57% hardware overhead, our solution is significantly more efficient.

References

[1] [n. d.]. AnyCore-riscv. https://github.com/anycore/anycore-riscv.
[2] [n. d.]. Championship Value Prediction 1 Secret Traces. https://perscido.univ-grenoble-alpes.fr/datasets/DS384.
[3] [n. d.]. ChampSim simulator. https://github.com/ChampSim/ChampSim.
[4] [n. d.]. Golden Cove microarchitecture. https://chipsandcheese.com/p/popping-the-hood-on-golden-cove.
[5] [n. d.]. IBPB. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-predictor-barrier.html.
[6] [n. d.]. IBRS. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-restricted-speculation.html.
[7] [n. d.]. IPC – The 1st Instruction Prefetching Championship. https://research.ece.ncsu.edu/ipc/.
[8] [n. d.]. SPEC2017 traces for ChampSim. https://dpc3.compas.cs.stonybrook.edu/champsim-traces/speccpu/.
[9] [n. d.]. STIBP. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/single-thread-indirect-branch-predictors.html.
[10] [n. d.]. Sunny Cove microarchitecture. https://chipsandcheese.com/p/sunny-cove-intels-lost-generation.
[11] [n. d.]. Type-Confusion. https://www.amd.com/system/files/documents/technical-guidance-for-mitigating-branch-type-confusion.pdf.
[12] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2006. Predicting secret keys via branch prediction. In *Topics in Cryptology–CT-RSA 2007: The Cryptographers' Track at the RSA Conference 2007, San Francisco, CA, USA, February 5-9, 2007. Proceedings*. Springer, 225–242.
[13] 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*. 312–320.
[14] Narasimha Adiga, James Bonanno, Adam Collura, Matthias Heizmann, Brian R Prasky, and Anthony Saporito. 2020. The ibm z15 high frequency mainframe branch predictor industrial product. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 27–39.
[15] Sam Ainsworth. 2021. Ghostminion: A strictness-ordered cache system for spectre mitigation. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 592–606.
[16] Truls Asheim, Boris Grot, and Rakesh Kumar. 2023. A storage-effective BTB organization for servers. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 1153–1167.
[17] Roberto Avanzi. 2017. The QARMA block cipher family. Almost MDS matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency s-boxes. *IACR Transactions on Symmetric Cryptology* (2017), 4–44.
[18] Grant Ayers, Jung Ho Ahn, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Memory hierarchy for web search. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 643–656.
[19] Mohammad Bakhshalipour, Seyedali Tabaeiaghdaei, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2019. Evaluation of hardware data prefetchers on server processors. *ACM Computing Surveys (CSUR)* 52, 3 (2019), 1–29.
[20] Rahul Bera, Konstantinos Kanellopoulos, Anant Nori, Taha Shahroodi, Sreenivas Subramoney, and Onur Mutlu. 2021. Pythia: A customizable hardware prefetching framework using online reinforcement learning. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 1121–1137.
[21] Ravi Bhargava and Kai Troester. 2024. AMD Next Generation" Zen 4" Core and 4 th Gen AMD EPYC™ Server CPUs. *IEEE Micro* (2024).
[22] Anubhav Bhatla, Biswabandan Panda, et al. 2024. The Maya Cache: A Storage-efficient and Secure Fully-associative Last-level Cache. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 32–44.
[23] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH computer architecture news* 39, 2 (2011), 1–7.
[24] Rahul Bodduna, Vinod Ganesan, Patanjali Slpsk, Kamakoti Veezhinathan, and Chester Rebeiro. 2020. Brutus: Refuting the security claims of the cache timing randomization countermeasure proposed in ceaser. *IEEE Computer Architecture Letters* 19, 1 (2020), 9–12.
[25] James Bonanno, Adam Collura, Daniel Lipetz, Ulrich Mayer, Brian Prasky, and Anthony Saporito. 2013. Two level bulk preload branch prediction. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 71–82.
[26] Joseph Bonneau and Ilya Mironov. 2006. Cache-collision timing attacks against AES. In *Cryptographic Hardware and Embedded Systems-CHES 2006: 8th International Workshop, Yokohama, Japan, October 10-13, 2006. Proceedings 8*. Springer, 201–215.
[27] Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knezevic, Lars R Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, et al. 2012. PRINCE–a low-latency block cipher for pervasive computing applications. In *Advances in Cryptology–ASIACRYPT 2012: 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings 18*. Springer, 208–225.

[28] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A systematic evaluation of transient execution attacks and defenses. In *28th USENIX Security Symposium (USENIX Security 19)*. 249–266.

[29] Rangeen Basu Roy Chowdhury, Anil K Kannepalli, Sungkwan Ku, and Eric Rotenberg. 2016. AnyCore: A synthesizable RTL model for exploring and fabricating adaptive superscalar cores. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 214–224.

[30] Brian W Curran, Christian Jacobi, JJ Bonanno, DA Schroter, KJ Alexander, A Puranik, and Markus M Helms. 2015. The IBM z13 multithreaded microprocessor. *IBM Journal of Research and Development* 59, 4/5 (2015), 1–1.

[31] Mark Evers, Leslie Barnes, and Mike Clark. 2022. The AMD next-generation "Zen 3" core. *IEEE Micro* 42, 3 (2022), 7–12.

[32] 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.

[33] Barry Fagin. 1997. Partial resolution in branch target buffers. *IEEE Trans. Comput.* 46, 10 (1997), 1142–1145.

[34] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. *Acm sigplan notices* 47, 4 (2012), 37–48.

[35] Brian Grayson, Jeff Rupley, Gerald Zuraski Zuraski, Eric Quinnell, Daniel A Jiménez, Tarun Nakra, Paul Kitchin, Ryan Hensley, Edward Brekelbaum, Vikas Sinha, et al. 2020. Evolution of the samsung exynos cpu microarchitecture. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 40–51.

[36] CRYPTREC Lightweight Cryptography Working Group et al. 2017. CRYPTREC Cryptographic Technology Guideline (Lightweight Cryptography). *CRYPTREC Report March* (2017).

[37] Vishal Gupta and Biswabandan Panda. 2022. Micro btb: A high performance and storage efficient last-level branch target buffer for servers. In *Proceedings of the 19th ACM International Conference on Computing Frontiers*. 12–20.

[38] Matthew R Guthaus, James E Stine, Samira Ataei, Brian Chen, Bin Wu, and Mehedi Sarwar. 2016. OpenRAM: An open-source memory compiler. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–6.

[39] Nikos Hardavellas, Ippokratis Pandis, Ryan Johnson, Naju Mancheril, Anastassia Ailamaki, and Babak Falsafi. 2007. Database servers on chip multiprocessors: Limitations and opportunities. In *Proceedings of the Biennial Conference on Innovative Data Systems Research*.

[40] Yasuo Ishii, Jaekyu Lee, Krishnendra Nathella, and Dam Sunwoo. 2021. Re-establishing fetch-directed instruction prefetching: An industry perspective. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 172–182.

[41] Christian Jacobi, Anthony Saporito, Martin Recktenwald, Aaron Tsai, Ulrich Mayer, M Helms, AB Collura, P-K Mak, Robert J Sonnelitter, Michael A Blake, et al. 2018. Design of the IBM z14 microprocessor. *IBM Journal of Research and Development* 62, 2/3 (2018), 8–1.

[42] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. In *Proceedings of the 42nd annual international symposium on computer architecture*. 158–169.

[43] Vladimir Kiriansky and Carl Waldspurger. 2018. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757* (2018).

[44] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2020. Spectre attacks: Exploiting speculative execution. *Commun. ACM* 63, 7 (2020), 93–101.

[45] 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)*.

[46] Rakesh Kumar and Boris Grot. 2022. Shooting down the server front-end bottleneck. *ACM Transactions on Computer Systems (TOCS)* 38, 3-4 (2022), 1–30.

[47] Amund Bergland Kvalsvik, Pavlos Aimoniotis, Stefanos Kaxiras, and Magnus Själander. 2023. Doppelganger loads: A safe, complexity-effective optimization for secure speculation schemes. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–13.

[48] Jaekyu Lee, Yasuo Ishii, and Dam Sunwoo. 2020. Securing branch predictors with two-level encryption. *ACM Transactions on Architecture and Code Optimization (TACO)* 17, 3 (2020), 1–25.

[49] Kevin Lim, Parthasarathy Ranganathan, Jichuan Chang, Chandrakant Patel, Trevor Mudge, and Steven Reinhardt. 2008. Understanding and designing new server architectures for emerging warehouse-computing environments. *ACM SIGARCH Computer Architecture News* 36, 3 (2008), 315–326.

[50] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *arXiv preprint arXiv:1801.01207* (2018).

[51] Yunzhe Liu, Xinyu Li, Tingting Zhang, Tianyi Liu, Qi Guo, Fuxin Zhang, and Jian Wang. 2024. AVM-BTB: Adaptive and Virtualized Multi-level Branch Target Buffer. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 17–31.

[52] Agustín Navarro-Torres, Biswabandan Panda, Jesús Alastruey-Benedé, Pablo Ibáñez, Víctor Viñals-Yúfera, and Alberto Ros. 2022. Berti: an accurate local-delta data prefetcher. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 975–991.

[53] Samuel Pakalapati and Biswabandan Panda. 2020. Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 118–131.

[54] Arthur Perais and Rami Sheikh. 2023. Branch Target Buffer Organizations. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 240–253.

[55] 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.

[56] 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.

[57] Moinuddin K Qureshi. 2019. New attacks and defense for encrypted-address cache. In *Proceedings of the 46th International Symposium on Computer Architecture*. 360–371.

[58] Glenn Reinman, Todd Austin, and Brad Calder. 1999. A scalable front-end architecture for fast instruction delivery. *ACM SIGARCH Computer Architecture News* 27, 2 (1999), 234–245.

[59] Glenn Reinman, Brad Calder, and Todd Austin. 1999. Fetch directed instruction prefetching. In *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 16–27.

[60] Glenn Reinman, Brad Calder, and Todd Austin. 1999. Fetch directed instruction prefetching. In *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 16–27.

[61] Gururaj Saileshwar and Moinuddin Qureshi. 2021. {MIRAGE}: Mitigating {Conflict-Based} Cache Attacks with a Practical {Fully-Associative} Design. In *30th USENIX Security Symposium (USENIX Security 21)*. 1379–1396.

[62] 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*. 73–86.

[63] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-privilege-boundary data sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 753–768.

[64] Niranjan K Soundararajan, Peter Braun, Tanvir Ahmed Khan, Baris Kasikci, Heiner Litz, and Sreenivas Subramoney. 2021. Pdede: Partitioned, deduplicated, delta branch target buffer. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 779–791.

[65] David Suggs, Mahesh Subramony, and Dan Bouvier. 2020. The amd "zen 2" processor. *IEEE micro* 40, 2 (2020), 45–52.

[66] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient {Out-of-Order} execution. In *27th USENIX Security Symposium (USENIX Security 18)*. 991–1008.

[67] Zhenghong Wang and Ruby B Lee. 2007. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th annual international symposium on Computer architecture*. 494–505.

[68] Zhenghong Wang and Ruby B Lee. 2008. A novel cache architecture with enhanced performance and security. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*. IEEE, 83–93.

[69] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F Wenisch, and Baris Kasikci. 2019. NDA: Preventing speculative execution attacks at their source. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 572–586.

[70] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. 2019. {ScatterCache}: thwarting cache attacks via cache set randomization. In *28th USENIX Security Symposium (USENIX Security 19)*. 675–692.

[71] Johannes Wikner, Daniël Trujillo, and Kaveh Razavi. 2023. Phantom: Exploiting decoder-detectable mispredictions. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 49–61.

[72] 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.

[73] Jiyong Yu, Trent Jaeger, and Christopher Wardlaw Fletcher. 2023. 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*. 1–14.

[74] Jiyong Yu, Namrata Mantri, Josep Torrellas, Adam Morrison, and Christopher W Fletcher. 2020. Speculative data-oblivious execution: Mobilizing safe prediction for safe and efficient speculative execution. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 707–720.

[75] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W Fletcher. 2019. Speculative taint tracking (stt) a comprehensive protection for speculatively accessed data. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 954–968.

[76] 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.

[77] Zhiyuan Zhang, Mingtian Tao, Sioli O'Connell, Chitchanok Chuengsatiansup, Daniel Genkin, and Yuval Yarom. 2023. {BunnyHop}: Exploiting the Instruction Prefetcher. In *32nd USENIX Security Symposium (USENIX Security 23)*. 7321–7337.

[78] 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.

[79] 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 *HPCA*. 346–359.