Securing Branch Predictors With Two-Level Encryption

JAEKYU LEE, Arm Research, USA
YASUO ISHII, Arm, USA
DAM SUNWOO, Arm Research, USA

Modern processors rely on various speculative mechanisms to meet performance demand. Branch predictors are one of the most important micro-architecture components to deliver performance. However, they have been under heavy scrutiny because of recent side-channel attacks. Branch predictors are indexed using the PC and recent branch histories. An adversary can manipulate these parameters to access and control the same branch predictor entry that a victim uses. Recent Spectre attacks exploit this to set up speculative-execution-based security attacks.

In this paper, we aim to mitigate branch predictor side-channels using two-level encryption. At the first level, we randomize the set-index by encrypting the PC using a per-context secret key. At the second level, we encrypt the data in each branch predictor entry. While periodic key changes make the branch predictor more secure, performance degradation can be significant. To alleviate performance degradation, we propose a practical set update mechanism that also considers parallelism in multi-banked branch predictors. We show that our mechanism exhibits only 1.0% and 0.2% performance degradation while changing keys every 10K and 50K cycles, respectively, which is much lower than other state-of-the-art approaches.

CCS Concepts: • **Security and privacy → Side-channel analysis and countermeasures**; • **Computer systems organization → Architectures**.

Additional Key Words and Phrases: Branch predictor, side-channel, encryption

ACM Reference Format:
Jaekyu Lee, Yasuo Ishii, and Dam Sunwoo. 2020. Securing Branch Predictors With Two-Level Encryption. *ACM Trans. Arch. Code Optim.* , (2020), 25 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

To meet the ever-increasing performance demand, modern processors rely on various speculative mechanisms, including branch prediction, hardware prefetching, dynamic cache insertion and replacement policies, and value prediction. These mechanisms use patterns from history to predict future behaviors. To improve prediction accuracy, finer-grained information is often used for training. In particular, we commonly use the instruction address, also known as the program counter (PC) or instruction pointer (IP), because the program behavior from dynamic instances of a given PC tend to be similar, while instructions associated with different PCs have distinct behaviors.

Most speculative mechanisms, however, have become an inadvertent source of information leakage, as seen from various side-channel attacks [9, 12, 18, 22, 23, 38, 45]. With the prevalence of public cloud computing, the operating system can be compromised in some hostile environments,

Authors' addresses: Jaekyu Lee, Arm Research, Building 1, 5707 Southwest Pkwy Suite 100, Austin, TX, USA, jaekyu.lee@arm.com; Yasuo Ishii, Arm, Building 1, 5707 Southwest Pkwy Suite 100, Austin, TX, USA, yasuo.ishii@arm.com; Dam Sunwoo, Arm Research, Building 1, 5707 Southwest Pkwy Suite 100, Austin, TX, USA, dam.sunwoo@arm.com.

XXXX-XXXX/2020/0-ART $15.00
https://doi.org/10.1145/1122445.1122456

which makes security attacks even easier. Branch predictors are one of the earliest discovered and frequently targeted side-channels [8, 9, 21, 22, 50]. They can be vulnerable because different contexts[1] share them within the same physical core. Branch predictors are typically indexed using the PC and recent branch histories. An adversary can control a branch predictor entry that a victim uses by deliberately manipulating the PC and the branch histories. Through this collision-based attack, the adversary can steer the control flow [20] or steal information from the branch predictor [9].

To make the branch predictor more secure, we can encrypt information using per-context keys. Because of the vulnerability of static encryption, as shown in [23], encryption keys should be changed periodically. Upon key changes, however, we lose all trained information unless (1) we re-encrypt all relevant entries with the new key, which can be very time-consuming, or (2) we can decrypt each piece of information with the exact old key, which may require a significant amount of book-keepings. Naïvely updating entries requires non-trivial hardware overhead and may increase the branch prediction latency. The branch predictor is one of the busiest micro-architecture units in the core pipeline because it can be speculatively accessed when an instruction is fetched [27]. Thus, we cannot afford any extra latency to branch prediction because of the timing criticality.

In this paper, we propose a two-level encryption scheme where both the index and the data (targets and directions) of the branch predictors are encrypted using separate schemes. If we encrypt the data only, collisions are still possible, and, thus, even if the key changes, the attacker can extract information and eventually break the encryption. If we encrypt the index only, manipulated collisions are less likely to occur, but we cannot use stronger encryption, as updating information upon key changes incurs considerable hardware and performance overhead. In our scheme, we use simple encryption at the first level indexing, thus allowing for efficient swap-based set updates, and we employ stronger data encryption at the second level to tighten security. To better utilize the branch predictor under dynamic encryption, we focus on *how to update all entries efficiently with a new key*. We also consider the bank-level-parallelism (BLP) in the branch predictor. Branch predictor tables are often implemented using multiple banks to accommodate many entries with multiple accesses in a cycle. The update process is a series of branch predictor read and write operations, so bank conflicts increase the update latency. To exploit the BLP, we guarantee that branch predictor sets from different banks will be swapped during the update process.

The **contributions** of our paper are as follows:

- We propose a two-level dynamic encryption mechanism to mitigate branch predictor side-channels. We randomize branch predictor accesses and encrypt the data stored in the table.
- We propose an efficient bank-level-parallelism-aware set update mechanism to update branch predictor entries upon key changes.
- We compare our mechanism against the Branch Retention Buffer (BRB) [43], the state-of-the-art branch predictor side-channel mitigation, and our mechanism significantly outperforms BRB with much smaller hardware overhead.
- Our update mechanism has negligible performance degradation while providing a more secure branch prediction even with a very short epoch length.

2 BACKGROUND

2.1 Branch Predictor

Branch predictors have been widely studied for decades. Recent processors have multiple branch prediction components providing prediction support for target, direction, indirect branch target, loop, and return address. The branch target buffer (BTB) is used for branch target prediction.

[1] A context can be a thread or a process, but we do not differentiate these unless noted.

Branch direction (taken or not taken) can be predicted using directional predictors [17, 25, 36, 48]. Each entry has a saturating counter, where the most significant bit (MSB) indicates the predicted direction. We store return addresses in the return address stack (RAS) on function calls. Upon a return instruction, which is an indirect jump, the return address is predicted by the top of the RAS. There exist predictors for loops and indirect branches, where multiple branch targets exist for a branch instruction. Figure 1 shows an overview of a typical modern branch predictor.

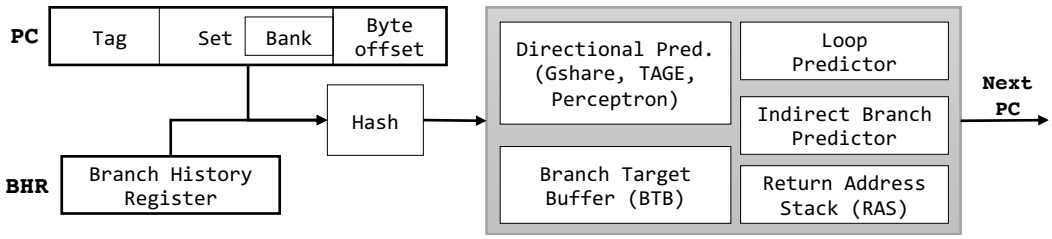


Fig. 1. Branch predictor overview.

To reduce prediction latency, we can access branch predictors at the fetch pipeline stage. Although the instruction type is not known during the fetch stage, and uncovered only after the decode stage, hitting in the BTB indicates that the fetched instruction is a branch instruction. We find out the target at the decode stage for direct branches and the execution stage for indirect branches. The branch direction will be resolved at the execution stage for conditional branches.

To index into these predictors, we mainly use the PC and the branch history register (BHR) that holds the recent global branch outcomes. For example, gshare [25] and TAGE [36] predictors use the XOR value of the PC and the BHR. A larger branch predictor is needed to cover larger instruction footprints. To support large structures with multiple accesses in a cycle, predictors are often implemented using multiple banks with the most cost-effective single-ported RAM [37]. As shown in Figure 1, the bank ID is usually acquired from the lower bits of the PC (after byte offset) to reduce bank conflicts in accesses from sequential PCs.

2.2 Branch Collision Attacks

Most side-channel attacks on the branch predictor exploit collisions. The branch predictor is typically shared by different contexts in the *same physical core*. As opposed to memory locations in data caches and main memory that can be protected by address translation mechanisms, branch predictor entries may not be protected for a context because branch predictor look-ups are performed *before* or *while* the instruction translation look-aside buffer (TLB) is accessed. Also, to reduce storage overhead, most structures store partial tags. As a result, an adversary can map its branch instruction into the branch predictor entry that the victim uses by manipulating both PCs and branch histories.

Figure 2 shows an example of a branch collision. The PCs of the two branch instructions (BEQ else) at 0x.....014 are almost identical except the two highest nibbles. Assuming the same branch history, these instructions will be mapped to the same branch predictor entry. We will detail other branch predictor side-channel attacks in Section 6.1.

2.3 Threat Model

In this work, we assume the following threat model:

- The branch predictor is shared by all contexts in the same physical core.
- Both victim and attacker applications reside in the same core but as separate processes. The attacker tries to infer prior branch outcomes and/or to change the control flow to its gadget code.

```
0x001...010:  CMP R1, #0
0x001...014:  BEQ else
0x001...xxx:  ADD R2, R2, #1
0x001...xxx:  B merge
            else:
0x001...030:  SUB R2, R2, #1
            merge:
0x001...xxx:  MOV R1, R2
```

(a) Victim

```
0xff1...010:  CMP R1, R1
0xff1...014:  BEQ else
0xff1...xxx:  NOP
0xff1...xxx:  B merge
            else:
0xff1...030:  NOP
            merge:
0xff1...xxx:  NOP
```

(b) Shadow

Fig. 2. Branch collision attack in [22].

- Some attacks rely on timing side-channel, i.e., measuring the timing difference of a certain event. For high-bandwidth side-channel attacks, slowing down the victim application may be necessary [4, 14, 18].

3 MITIGATING SIDE-CHANNEL ATTACKS TWO-LEVEL ENCRYPTION

We now describe our two-level dynamic encryption mechanism to mitigate branch predictor side-channels. To make a prediction, we first index (hash) into the predictor and then use data stored in the table. The set-index and data are not orthogonal since the index function determines which entry (data) to use. We aim to protect both index and data using encryption. We perform periodic key updates to overcome the weakness of static encryption [23], which makes it necessary to re-index the sets upon key changes. To tailor-make such a set update mechanism for the branch predictor, we use exclusive-or (XOR)-based encryption for the set-indexing (§3.4). To further enhance the security, we use stronger encryption for the data (§3.5) stored in the predictors.

3.1 How Two-Level Encryption Mitigate Side-channel Attacks

We use the PC to access all branch predictor tables. Encrypting the PC with a per-context secret key provides index randomization for set-associative structures. Then, second-level data encryption further improves security. Figure 3 shows an example of how encryption can prevent branch collisions for the BTB. Without encryption, branch collisions can be easily created by setting similar PCs (case 1). With encryption using secret keys per context, set indices are randomized, and the target information is encrypted (case 2). It becomes very difficult to create collisions and, even if an adversary manages to map to the same entry, the prediction information in the table cannot be correctly decrypted without knowing the key.

```
// Instruction 1 (Process 0)
pc1 = 0x80d12054
target1 = 0x80d12064
key1 = 0x7f40f

// Instruction 2 (Process 1)
pc2 = 0x40d12054
target2 = 0x40d12080
key2 = 0x1c4a

// Set ID: PC[14:5], 10-bit
// Targ: Target[20:0], 21-bit
```

```
// Case 1. No encryption
set1 = (pc1>>5) & 0x3ff = 258
set2 = (pc2>>5) & 0x3ff = 258        Set collision!
targ1 = target1 & 0x1fffff = 0x112064
targ2 = target2 & 0x1fffff = 0x112080

// Case 2. XOR encryption
set1 = ((pc1>>5)^key1) & 0x3ff = 269
set2 = ((pc2>>5)^key2) & 0x3ff = 328
targ1 = (target1^key1) & 0x1fffff = 0x16d46b
targ2 = (target2^key2) & 0x1fffff = 0x113cca
                    No collision! Target encrypted as well.
```

Fig. 3. BTB encryption example.

We change the encryption key for each context once an epoch expires. An *epoch* is an execution period during which the same encryption key, but different from other contexts, is used for a context. The length of an epoch can be determined based on *time-based* (cycle counts or the number of branch instructions) or *pre-defined events*, including context switches and interrupts. These two types of epochs are not mutually exclusive, and it would be better to use together. An epoch transition takes place when a timer expires or a pre-defined event occurs. Also, an epoch is *independently* maintained per context, and the length of epoch has to be set to balance between security risk and performance impact. If an epoch is too short, we may lose trained information more frequently, thereby degrading performance. On the other hand, if an epoch is too long, the risk of a security breach increases.

3.2 Limitation of Adopting CEASER for Branch Predictor

The fundamental idea of our index encryption mechanism and dynamic remapping resembles that of CEASER [31, 32], which encrypts physical line addresses so that address-to-set mapping becomes more difficult to identify. It performs periodic encryption key changes with dynamic remapping. CEASER is effective in preventing conflict-based timing attacks, but it has two limitations that prohibit its adoption for branch predictors.

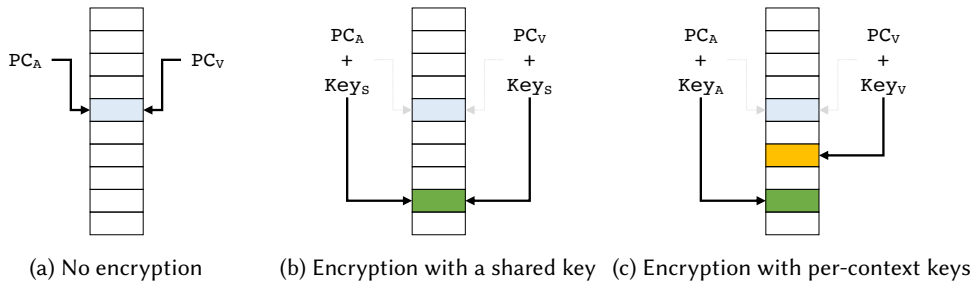


Fig. 4. Address-to-set mapping example (A: attacker, V: victim, S: shared key).

The first limitation is that encryption with a shared key cannot prevent branch collisions. For example, in Figure 4, knowing the victim's virtual address (PC_V), the attacker can manipulate its address (PC_A) similarly. Without encryption (4a) and with the shared key encryption (4b), although the mapping leads to different sets, addresses from the attacker and the victim collide in the same entry. We overcome this limitation by maintaining per-context encryption keys (4c).

The second limitation comes from its update algorithm. CEASER uses incremental updates by ascending set ID order. Cache lines in a set can be remapped to an arbitrary set. If a remapped line incurs a cache eviction, we need to handle this line. We can stage the line in a temporary buffer and perform another remapping. However, this may subsequently evict another line. We call this a *chained set update*. Figure 5 shows the pseudo-code of a chained update algorithm. Note that we assume a direct-mapped structure for simplicity. The chained update makes *predeterminedly* knowing which sets have been updated almost impossible because the order of updated sets is very random.[2] One series of chain update will complete if NewSet (line 20) is previously updated. Then, we can start a new series from the next not-yet-updated set (lines 10-15). This method may require a 1-bit flag for each of the branch predictor sets (UpdateDone) to indicate that the corresponding

[2]This randomness is what many encryption mechanisms aim for by design.

```
// initialization
for (idx = 0; idx < NumSet; ++idx)
  UpdateDone[idx] = false;

OldSet = 0;
Read(Buffer[0], Table[OldSet]);
while (true) {
  if (UpdateDone[OldSet]) {
    // find the next set to update
    for (idx = 0; idx < NumSet; ++idx) {
      if (!UpdateDone[idx]) {
        OldSet = idx;
        break;
      }
    }
    if (UpdateDone[OldSet]) // all sets updated
      break;
  }
  // read entries from the new set
  NewSet = GetNewSet(OldSet, OldKey, NewKey);
  Read(Buffer[1], Table[NewSet]);
  // update and write the entry into new set
  Update(Buffer[0]);
  Write(Buffer[0], Table[NewSet]);
  // prepare for next iteration
  Swap(Buffer[0], Buffer[1]);
  OldSet = NewSet;
}
```

Fig. 5. Pseudo-code of a (chained) set update mechanism.

set has been updated using a new encryption key for the current epoch. We need to check this large per-set structure upon every predictor access, which incurs extra cycles to branch prediction latencies. While this latency may be tolerable in the last-level cache, where CEASER was intended to be used, the branch predictor is on a very timing-critical path, thus, adding extra latency to the critical path will degrade the performance significantly. Also, this storage is not scalable with a larger branch predictor.

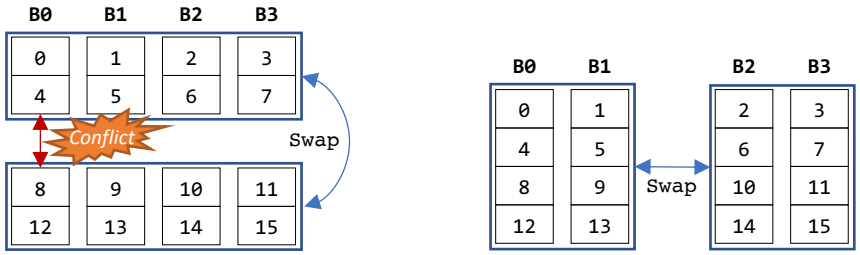
3.3 BLP-aware Light-weight Set Update

The main problem of the naïve update mechanism in Figure 5 is that it requires a large per-set structure (UpdateDone). To remove this structure, the identification of updated sets should be *predetermined* using a set ID and time information. For the branch predictor, a swap-based update is preferable, i.e., OldSet and NewSet in Figure 5 are mutually exchanged during the update process. In particular, we use XOR-based encryption, which is commonly used in error detection and simple encryption. The XOR encryption enables a swap-based update because of its commutative and associative properties (Equations 1-7). OldSet (Set_t in Eq. 2 and Eq. 7) and NewSet (Set_{t+1} in Eq. 3 and Eq. 6) can be computed by simply XOR-ing with $SwapKey$ (Eq. 8). Note that the following equations are used for a context, where t and $t+1$ are adjacent epochs, BHR is the content of the branch history register, Set is the set id that PC is mapped, and Key is an encryption key. Each context maintains its own encryption keys.

$$Index = PC[n1 : n2] \oplus BHR \tag{1}$$
$$Set_t = Index \oplus Key_t \tag{2}$$
$$Set_{t+1} = Index \oplus Key_{t+1} \tag{3}$$
$$Index = Set_t \oplus Key_t \tag{4}$$
$$Set_{t+1} = (Set_t \oplus Key_t) \oplus Key_{t+1} \tag{5}$$
$$Set_{t+1} = Set_t \oplus (Key_t \oplus Key_{t+1}) \tag{6}$$
$$Set_t = Set_{i+1} \oplus (Key_t \oplus Key_{t+1}) \tag{7}$$
$$SwapKey = (Key_t \oplus Key_{t+1}) \tag{8}$$

The XOR-based encryption, however, does not automatically provide a predetermined update. To provide predeterminism, we split an epoch into multiple sub-epochs and swap set t and its complement set ($t \oplus$ SwapKey) in each sub-epoch t. However, naïvely setting SwapKey is problematic. For example, if SwapKey is 0x4, as shown below, the swap in sub-epoch 4 will negate the update performed at sub-epoch 0. As a result, no sets will be properly updated.

- Sub-epoch 0: swap sets 0 and 4 ($0 \oplus 4$)
- Sub-epoch 1: swap sets 1 and 5 ($1 \oplus 4$)
- ...
- Sub-epoch 4: swap sets 4 and 0 ($4 \oplus 4$)



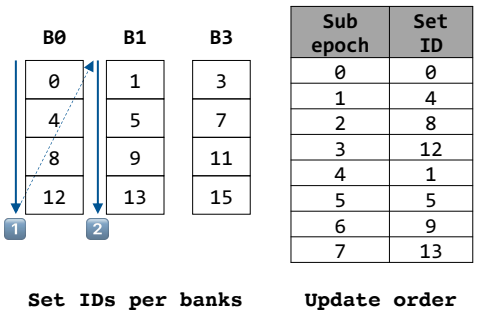
(a) Set MSB swap (b) Bank MSB swap

Fig. 6. BLP-aware swap-based set update.

To avoid this problem, we can always set the most significant bit (MSB) of SwapKey so that the MSB of set IDs to swap will be different. However, this method cannot fully utilize bank-level-parallelism (BLP). As explained in Section 2.1, the bank ID is acquired from lower PC bits. If we use the set-MSB to swap, then bank conflicts may occur with a probability of $\frac{1}{NumBank}$. For example, assuming 16 entries with 4 banks, sets 0, 4, 8, and 12 belong to bank 0. In Figure 6a, the set-MSB swap maps set 0 to any set from 8 to 15. Mapping to either set 8 or 12 will cause bank conflicts. To utilize the BLP fully, we always set the bank-MSB (the MSB of the bank ID) of SwapKey to 1 (Eq. 13), which is shown in Figure 6b. To do so, we need to alternate the bank-MSB of encryption keys in every epoch by clearing it in even-numbered epochs (Eq. 11) and setting it in odd-numbered epochs (Eq. 12).

$$BankSet = NumBank >> 1 \tag{9}$$
$$BankClear = (NumSet - 1) \oplus BankSet \tag{10}$$
$$Set_{even\ t} = Index \oplus (Key_t \& BankClear) \tag{11}$$
$$Set_{odd\ t} = Index \oplus (Key_t \mid BankSet) \tag{12}$$
$$SwapKey = (Key_t \oplus Key_{t+1}) \mid BankSet \tag{13}$$

This bank-MSB swapping requires subtle changes in the sub-epoch mechanism. To provide predeterminism while exploiting BLP, we want to iterate over all sets in banks 0 and 1. The order of sets being updated will be 0, 4, 8, 12, 1, 5, 9, and 13, as seen in Figure 7a, which does not match with simply ascending sub-epoch IDs. As a result, we need to translate sub-epoch IDs to actual set IDs. Figure 7b shows a pseudo-code of the translation.



(a) Sub-epoch and the order of updated sets.

```
1 int set_per_bank = NumSet / NumBank;
2 int SetToSEpoch(int set) {
3   int bank = set & (NumBank - 1);
4   return bank * set_per_bank +
        (set>>log2(NumBank));
5 }
6 int SEpochToSet(int sub_epoch) {
7   int bank = (int)(sub_epoch / set_per_bank);
8   return (sub_epoch % set_per_bank) * NumBank
        + bank;
9 }
```

(b) Translation between sub-epoch and set IDs.

Fig. 7. The order of updated sets and translation between sub-epoch and set IDs.

Figure 8 shows an example of the swap-based set update with 4 banks using the bank-MSB. In epoch 0, the encryption key b11**0**1 (note that bit[1] is the bank-MSB and is cleared in even-numbered epochs) is XOR-ed with set IDs. Sets 0 and 1 (highlighted) are mapped within the first-half banks. In epoch 1, the key is b10**1**1 and the SwapKey b01**1**0 (= b1101 ⊕ b1011) is used to swap. As a result, sets 0, 6 and 1, 7 are swapped (0 ⊕ 6 = 6, 1 ⊕ 6 = 7).

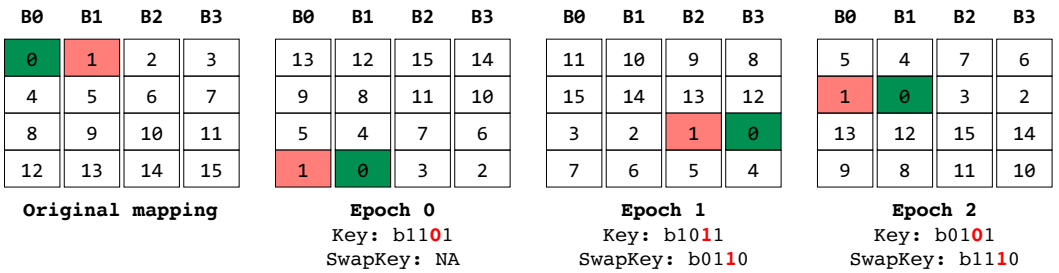


Fig. 8. Swap-based set update example.

3.4 Index Encryption: Putting It All Together

Figure 9a shows the pseudo-code of our BLP-aware set update mechanism (BSUP). To swap sets, a sub-epoch goes through three phases: 1) read entries from two sets into the temporary buffer, 2) perform re-encryption, and 3) write all entries back to the table. The simplest way to perform an update is to block accesses to the corresponding two banks and quickly update all entries. However, any branch prediction from these banks will not be available during the update process, which will negatively affect performance. Instead, we can opportunistically read and write entries based on the *bank availability*. Consequently, read and write phases have variable latencies based on bank access

patterns. Once all three phases complete, we move on to the next sub-epoch. A complete update requires $\frac{\text{NumSet}}{2}$ sub-epochs. During an update, accesses to the set can still be serviceable either from the main array or the temporary buffer. The arbitration logic of the branch predictor requires changes accordingly, but the overhead will be trivial. A potential weakness of this mechanism is that if an epoch transition occurs before all sets are updated because of either short epoch lengths or epoch-terminating events, we will lose information from sets that are not updated. However, this drawback applies to any encryption mechanism with very short epochs.

```
for (SubEpoch=0; SubEpoch<NumSet/2;
     ++SubEpoch) {
  // real set id
  OldSet = SEpochToSet(SubEpoch);
  // complement set
  NewSet = OldSet ^ SwapKey;

  // Phase 1: Read
  Read(Buffer[0], Table[OldSet]);
  Read(Buffer[1], Table[NewSet]);

  // Phase 2: Update
  Update(Buffer[0]);
  Update(Buffer[1]);

  // Phase 3: Write
  Write(Buffer[0], Table[NewSet]);
  Write(Buffer[1], Table[OldSet]);
}
```

(a) Set update mechanism

```
int Index(int SetID, int SubEpoch) {
  // alternate the bank MSB
  if (EpochID % 2 == 1) {
    OldKey = Key[Epoch-1] & BankClear;
    NewKey = Key[Epoch] | BankSet;
  } else {
    OldKey = Key[Epoch-1] | BankSet;
    NewKey = Key[Epoch] & BankClear;
  }

  OldSet = SetID ^ OldKey; // set IDs
  NewSet = SetID ^ NewKey;
  // sub-epoch IDs
  OldSE = SetToSEpoch(OldSet);
  NewSE = SetToSEpoch(NewSet);
  SID = (OldSE >= (NumSet/2)) ? NewSE:
        OldSE;
  return SID < SubEpoch ? NewSet: OldSet;
}
```

(b) Set-index mechanism

Fig. 9. Sub-epoch based index encryption mechanism.

Figure 9b shows the set index mechanism. There are two set index candidates, one with the old key (OldSet) and the other with the new key (NewSet). We can identify whether a set was updated by comparing the set ID and the current sub-epoch ID. We first translate the set ID to the sub-epoch ID. If the translated sub-epoch ID (OldSE) is greater than the maximum sub-epoch ID ($\frac{\text{NumSet}}{2} - 1$), we use its complement ID (NewSE) instead. If the final ID (SID) is less than current sub-epoch ID, this set has been updated already, and we return NewSet. Otherwise, OldSet will be returned.

BSUP needs the following new structures and hardware changes.

- Random number generator (RNG): The success of any encryption relies heavily on the strong random number generation, and most of the recent processors include stronger random number generators [5, 7] for cryptography. Note that acquiring a random number from a hardware RNG takes 110 ns in Intel's Kaby Lake-S microarchitecture [10]. The operating system maintains random number generations and 110 ns is short enough not to require buffering random numbers.
- Key table: We use a 64-bit key and need to store two keys (previous and current) per context. Keys are not directly readable by the context and not visible to other contexts. We need a 10-bit index to access predictors, so we use XOR folding (e.g., key[0:9] $\oplus$ key[10:19] $\oplus$ key[20:29]). These keys are used for data encryption in Section 3.5 as well.
- Epoch ID: We need to identify parity (even or odd) of epochs to alternate the bank-MSB for each context.

- Sub-epoch ID: We need to maintain a sub-epoch ID for each context. The sub-epoch ID will be reset at the beginning of a new epoch, and the maximum value of sub-epoch ID is NumSet/2.
- Sub-epoch state: This indicates the current phase of a sub-epoch (0: read, 1: update, 2: write, 3: done).
- Buffer: We need storage for two sets for each predictor tables (target and direction predictors).

Table 1 summarizes the hardware overhead of BSUP. We need to maintain the key table for the maximum number of concurrent contexts, but other structures are for each simultaneous multi-threading (SMT) threads in a core. If we support up to 32 contexts per core, it requires 654 bytes, which is 1.2% of the total branch predictor budget.

Table 1. The hardware overhead of BSUP.

Structure	Size	Context	Thread
Key table	64b x 2	✓	
Epoch id	1b		✓
Sub-epoch id	10b		✓
Sub-epoch state	2b		✓
BTB sets	6B x 4way x 2		✓
Bimodal sets	1B x 2		✓
TAGE sets	3B x 2 x 15		✓
Total		512B (up to 32 contexts) per core	141.7B per SMT thread

3.5 Data Encryption

XOR encryption does not guarantee strong protection in general. However, it enables a very efficient set update mechanism to cope with dynamic key changes for indexing. To further improve security, we add second-layer protection on the prediction data such as targets and prediction counters. Note that tags can also be encrypted, but we do not consider it in this paper because of significant changes in tag lookup logic. Unlike indexing that requires 1-cycle decryption, we can use a stronger encryption algorithm for data because it needs to be decrypted at the end of the 3-stage branch prediction pipeline. In other words, we can tolerate up to 2-cycle decryption latency for data. A lightweight block-cipher (80-bit of storage and less than 2 clock cycles latency) was proposed in CEASER [31] to encrypt 40-bit physical line address. A similar encryption scheme can be used. Branch target encryption [13] can be done straightforwardly. We encrypt target upon update and decrypt it before use. On the other hand, prediction counter update requires 1) read, 2) decrypt, 3) update, 4) encrypt, and 5) write.

Figure 10 shows a schematic diagram of the branch predictor encryption. At cycle 0, we access both BTB and TAGE predictors in parallel. The outcome from the TAGE predictor will be available at cycle 2. Based on which level of the BTB the access hits in, the target will be available either at the end of cycle 0 (L1) or at the end of cycle 1 (L2/3). Branches predicted *taken* will redirect the next PC to the branch target at the end of cycle 3. Because we have two cycles to decrypt the target, as far as the decryption latency is within two cycles, our scheme will not incur additional latency.

3.6 Security Analysis

In this section, we analyze how our mechanism improves the security of a branch predictor. We will first categorize various attacks types, describe how one can perform attacks, and provide analysis.

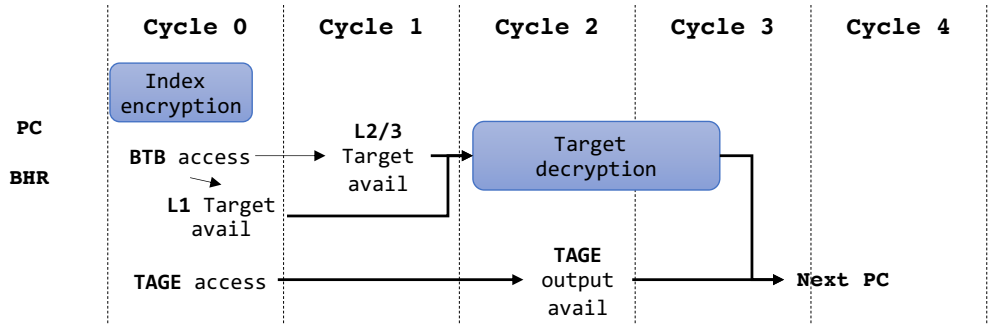


Fig. 10. Branch predictor encryption schematic diagram.

3.6.1 Branch Predictor Side-channel Attack Category. We categorize existing branch predictor attacks into three groups.

Category 1 To infer control flow graph, earlier attacks against RSA [1–3] tried to identify which branch predictor entry was accessed by the victim. These attacks require *sequential traversal* of the branch predictor. Since the adversary knows the victim's virtual addresses, it becomes trivial to infer the control flow.

Category 2 This category is similar to Category 1. BranchScope attack [9] extracts the branch outcome of a *specific entry* using a fine-grained Prime+Probe attack [26, 28, 41]. This requires pin-pointed access to the entry.

Category 3 Steal control flow under speculative execution: As seen from Spectre v1 (branch direction) and v2 (branch target) attacks [20], tainting branch predictors allows undesired and wrong code sections to access privileged information. Then, another side-channel such as a cache can complete the attack. This attack does not require a sophisticated method since an attacker can blindly pollute all branch predictor entries.

3.6.2 How to identify branch outcome? We can directly read branch information if a processor supports such capability, including Intel's LBR (last branch record) [16]. The LBR records 1) the address of a branch instruction, 2) the target address, 3) the prediction information whether the direction or target was mispredicted, and 4) the elapsed cycles between the branches. This can be useful to 1) identify delinquent branches and 2) re-construct the control flow information. However, the LBR has two limitations: 1) information is recorded for taken branches only and 2) this does not provide which set was mapped from a PC, which is essential to create collisions.

On the other hand, timing attacks such as Prime+Probe [26, 28, 41] and Flush+Reload [14, 47] have been used to infer the address-to-set mapping for branch predictors [1–3]. These methods rely on sequentially searching table structures to measure differences in access (execution) latency. For example, from Prime+Probe cache attack, if a victim evicts a cache line that belongs to the attacker, this results in a cache miss and a longer access latency. From the sequential search, we identify that the N-th entry or set was accessed by the victim. As pointed out in [31], the encryption mechanism improves security by obfuscating this sequential searching, so it makes the address-to-set mapping more difficult to identify.

3.6.3 Eviction Set-based Analysis. As pointed out in [23, 31], attacks against the last-level cache (LLC) are more challenging than against the L1 cache (and core structures) because of following reasons: (1) probing the LLC is much slower than the L1 cache because of 1) larger size, 2) higher associativity, and 3) longer latency and (2) while the L1 cache is virtually indexed, the LLC is

physically indexed. Attackers have full control of virtual addresses, so it is trivial to control accesses for the L1 cache. To overcome these challenges, eviction set-based attack was proposed [23]. Instead of identifying a set-mapping function, this attack formed a random conflict set of which size is greater than the associativity of the LLC and examined whether a cache miss occurs from any lines after repetitive accesses. If an access to any line takes longer than other lines, the corresponding line belongs to a conflict set. After some trials, the attack can create an eviction set. We perform a similar security analysis done in [31] for the branch predictor using the following equations, where N is the number of cache lines in the LLC, F is the fraction of cache lines, and R is the number of trial rounds for reliable attacks.

$$Num\_aceesses\_to\_form\_an\_eviction\_set = R \times L^2 \tag{14}$$

$$L = F \times N \tag{15}$$

$$P(bin\_has\_B\_balls) = \frac{e^{-\lambda} \cdot \lambda^B}{B!} \tag{16}$$

Using a bin-and-ball analysis (Eq. 16), an attack against a 16-way, 8MB LLC requires 42% of cache lines to be accessed ($F = 0.42$) and needs ~38000 accesses to create an eviction set.[3] We applied a similar analysis on the branch target buffer. With a 4-way, 4K-entry BTB, we identify that $F = 0.19$ and only ~300 accesses are necessary to create an eviction set.[4] Equations 14-16 show that the security of a structure is proportional to the size and the associativity of the structure. For branch predictors with smaller sizes and associativities than the LLC (number of sets: 1/8, associativity: 1/4), attacks against the branch predictor take 128x fewer accesses. Considering the shorter access latency of the branch predictor, the effective time to create an eviction set will be three orders of magnitude less.

3.6.4 Revisiting the Limitation of Encryption with Shared Key. When the index function is encrypted, it obfuscates *sequential traversal* of a structure, identifying the address-to-set mapping becomes more difficult (§3.6.3). However, as explained in Section 3.2, if the attacker and the victim share the same key to encrypt information, the branch predictor is still vulnerable to side-channel attacks. For Category 1 attacks, since *sequential traversals* are not possible, we can use an eviction-set, and the attack needs only ~300 accesses. For Category 2 and Category 3 attacks, the attacker can immediately create collisions once virtual addresses are known. Therefore, we can reaffirm the weakness of shared-key encryption mechanisms.

3.6.5 Security Analysis of BSUP. Maintaining separate encryption keys for different contexts can further improve the security of the branch predictor. Creating an eviction set (§3.6.3) becomes more complicated. As shown in Figure 4c, an eviction set from one context does not guarantee evictions for another context because of the different keys and mappings. As a result, without knowing own and victim's keys, Category 2 attacks become infeasible to exercise. However, an attacker can still perform Category 3 attacks. To blindly poison (brute-force) all branch predictor entries, it requires as many accesses as the number of entries.[5] For our baseline configuration in Table 3, the BTB and TAGE predictors require 4096 and 1024 accesses, respectively. TAGE predictor [36] comprises multiple tables, but the attack needs to taint the default bimodal predictor only.

Based on this observation, the index encryption alone is still not secure enough, and data encryption can boost security. The BTB stores 47-bit branch targets. With the lack of encryption

[3]With a 16-way, 8MB LLC, we use R=2, N=8192, F=0.42, and B=17.
[4]With a 4-way, 4K-entry BTB, we use R=2, N=4096, F=0.19, and B=5.
[5]Note that although we provide the upper bound analysis, the attack may need fewer accesses.

Table 2. Security analysis (✓: protected, ✗: not protected).

Scheme	Category 1	Category 2	Category 3 (Spectre v1)	Category 3 (Spectre v2)
No encryption	✗	✗	✗	✗
Shared-key encryption [31]	✓	✗	✗	✗
Index encryption (different keys)	✓	✓	1024	4096
BSUP	✓	✓	16384	✓

key identification, it is not practical to reverse-engineer the BTB encryption. Thus, Spectre v2 attacks can be mitigated. For Spectre v1 attacks, we encrypt 3-bit saturating counters in the TAGE predictor. Since there are only 8 possible counter values, the attacker can assume 1 of 8 values and train the TAGE table entries to have such value. To set a certain counter value, it requires two accesses on average. Consequently, data encryption on TAGE demands 16 times more accesses to taint. We summarized the security analysis in Table 2. BSUP is effective for Category 1, Category 2, and Spectre v2 attacks. Spectre v1 attacks require 16384 branch predictor accesses, which may not be secure enough. This brute-force attack can hardly be protectable except by partitioning the branch predictor, which requires a significant performance overhead. However, the attack can be less practical because branch targets are always mis-predicted under data encryption, which lengthens the attack duration.[6] For stronger protection, we can combine other side-channel mitigation schemes since Spectre v1 attacks must rely on additional side channels such as caches.

3.6.6 *Other Discussions.*

(1) Simultaneous Multi-Threading (SMT): In SMT processors, we need a selective update since multiple threads can be running simultaneously in a core. We can embed full context IDs (8 to 12 bits) to differentiate entries, which provides strong isolation among contexts. However, the hardware overhead is too significant to be implemented. Using partial information will weaken isolation. During an update for a context, we may encounter a mismatched number of entries that belong to the context from two sets to swap. In this case, we can 1) discard own surplus entries, which hurts its performance, or 2) evict entries from other contexts, which affects the performance of other applications. We will leave this as future work.

(2) Based on how we define a *context*, protection coverage and performance impact may vary. For example, if we define a context as a process, and if the victim and the attacker are in the same process, we cannot protect the branch predictor since they use the same encryption key. However, if we define a context as a thread, even if the victim and the adversary are in the same process, as far as they are separate threads, our protection scheme works. In this case, benign multi-threaded applications will be impacted by halving the branch predictor resources even if threads may have similar behaviors. If there are some Trojan gadget code within the same thread, no protection can work. To consider this, there is an effort to isolate each website in web browsers [33]. If pages originate from different websites, instead of allowing them to share some hardware resources, putting them in different processes gives natural protection.

(3) Upon context switches, we leave branch predictor entries intact, but the attacker cannot exploit them unless they know both their own key and the victim's key, as explained in

[6]The target mis-prediction penalty will be proportional to the front-end pipeline depth (typically 5-6 cycles, but can be 10 cycles in a deeper microarchitecture). If we assume 20% of branches are taken, 20% x 5 cycles will be added to the CPI stack. If an ideal CPI is ~0.5 (IPC=2.0), the CPI will become 1.5.

Section 3.6.2. Once the process is scheduled again, it can use entries using the old encryption key and perform remapping with the new key.

4 EXPERIMENTAL METHODOLOGY

4.1 Simulator

We use a proprietary cycle-accurate execution-driven Arm AArch64 simulator for evaluation. The simulator is used for CPU development and has been correlated against commercial CPUs. Table 3 shows the processor configuration, which is intended to be generic.

Table 3. Processor configuration.

Core	3 GHz, 8-wide
Branch predictor	TAGE [36]: 15 1024-entry direct-mapped tables, 4-bank
BTB	L1: 64 entry, fully associative L2: 256 entry, fully associative L3: 4096 entry, 4-way, 4-bank
Cache	L1: 64KB, 4-way, 4-cycle L2: 1MB, 8-way, 11-cycle L3: 8MB, 16-way, 34-cycle
Memory	90ns latency, 48GB/s bandwidth

4.2 Benchmarks and Evaluation Metric

We use SimPoints [29] with 30 million instruction intervals generated from single-threaded benchmarks from SPEC 2000, 2006, and 2017 suites [39] and GeekBench [30] (a total of 102 applications). We use the projected instruction per cycle (IPC) from weighted SimPoints as the performance evaluation metric.

5 RESULTS

In this section, we evaluate the performance impact of the different mechanisms upon key changes:

- Reindex: Index is encrypted, but data in predictors is *not encrypted*. Leave stale information in the table upon key changes.
- RI+Enc: Both index and data are *encrypted*. Leave encrypted information in the table upon key changes.
- Reset: Reset all entries at the end of each epoch, which is similar to Time protection [11].
- BRB: No encryption. Although the original BRB mechanism [43] replicates only directional predictors (bimodal and TAGE), we can apply a retention buffer to the BTB as well. As shown in Table 3, our configuration has 3 levels of BTBs, and we replicate the L1 and L2 BTBs for each context, where one replication requires 1.88KB (6B x (64 + 256) entries). For TAGE, the bimodal predictor and the TAGE table with the shortest history are replicated. This requires 4KB per context (bimodal: 1B x 1024 entries, TAGE: 3B x 1024 entries). The original BRB implementation uses Perceptron [17] as the default predictor, but we keep bimodal as in TAGE implementation instead. BRB resets shared predictors upon context switches. A length of epoch emulates that of context switches.
- BSUP: Our BLP-aware set update mechanism. Both index and data are *encrypted*. Update all entries with the new key upon key changes.

Note that we do not assume any specific epoch length, but try to show the performance impact of different epoch lengths. OS time slice can vary from a few ms to 10s of ms. As we explained in Section 3.1, the epoch should terminate on pre-defined events as well as branch predictor activities. As a result, we may perform multiple updates even within an OS time slice. We use cycle-based epochs and sweep the length of epochs from 5000 to 1 million cycles. We do not simulate context switches. The side-effect of context switches cannot be alleviated, except by resource partitioning. We estimate the performance impact of context switches by varying epoch lengths.

5.1 BTB Results

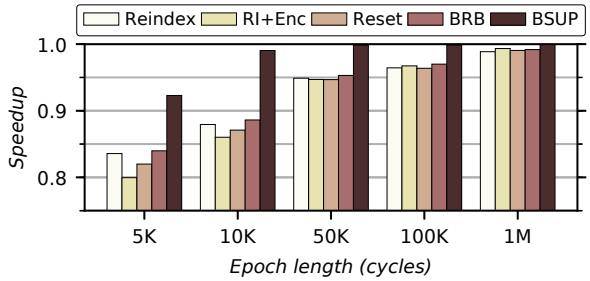
In this section, we evaluate how the dynamic encryption of the BTB affects performance. As explained in Section 2.1, the BTB is used for the branch target prediction, and it initiates early branch prediction even before branch instructions are decoded. Consequently, penalizing the BTB from the loss of prediction information can result in significant performance degradation. We evaluate mechanisms on the L3 BTB while the L1 and the L2 BTBs are reset except for BRB.

Figure 11 shows the results. The main difference between Reindex and RI+Enc is that if encryption keys within a short period map a PC to the same entry and the prediction information is still correct, Reindex can reuse this entry. However, even though there is a matching entry, the reused entry in RI+Enc will lead to a misprediction because of the encrypted target. RI+Enc and Reset show similar results. However, RI+Enc may lead to more mispredictions (hit in the BTB, but content decrypted with the wrong key), while Reset leads to no predictions (miss in the BTB) until re-trained. Figure 11a shows that Reindex outperforms both RI+Enc and Reset across all epochs despite security concerns. RI+Enc shows interesting results compared to Reset. Both schemes lose prediction information, but RI+Enc has more mispredictions instead of no predictions than in Reset. BTB mispredictions lead to pipeline flushes and have a greater penalty than not making predictions. BRB keeps the L1 and L2 BTB entries, which can provide predictions after key changes. However, its performance improvement over Reset is marginal, 2% and 1.5% improvement for 5K and 10K-cycle epochs, respectively. The benefit of retaining tagged structures like the BTB can be less effective than tag-less structures like the bimodal predictor because of the coverage. Tagged structures can greatly improve the prediction accuracy by tagging an entry with fine-grain information, but tag-less structures provide predictions for any access to the entry. Also, the target prediction has to be 100% accurate, but the direction prediction has a 50% chance of being correct for an unbiased branch. As the epoch length increases, we can amortize the performance penalty of losing trained information.

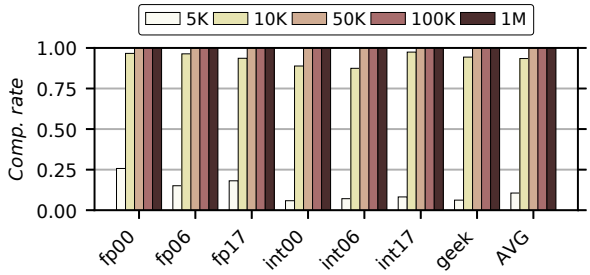
On the other hand, BSUP outperforms all schemes across all epoch lengths. Other than the very short 5K-cycle epoch, BSUP shows almost no performance degradation. We observe 7.7% performance degradation with 5K-cycle epochs. This is because the latency required to update all sets is usually longer than 5000 cycles, so not all sets are being updated. Note that the minimum update latency is 4608 cycles[7] and the bank conflicts will increase the latency. Figure 11b shows that the set update is completed for only 10-20% of all epochs in 5K-cycle epoch, thus we lose part of the trained information. The average update latency in Figure 11c explains the low update completion rate. The latency is longer than 5000 cycles in all suites, and 14 (out of 102) benchmarks show longer than 10000-cycle latencies.[8] As a result, it is almost impossible to complete the set update within 5K-cycle epochs for most benchmarks and 10K-cycle epochs are not sufficient for 13.7% of all benchmarks. Note that 500.perlbench shows the worst 34.4% performance degradation

[7]The update process for a sub-epoch requires 9 cycles for 4-way associative BTB (4 cycles: read, 1 cycle: update, 4 cycles: write) and there are 512 (NumSet/2) sub-epochs.

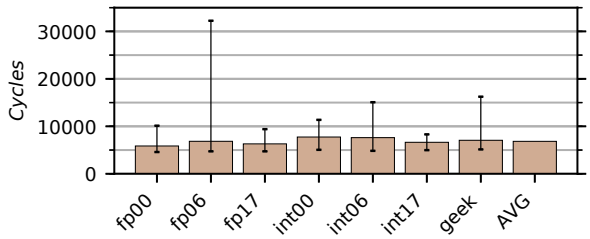
[8]Branch-intensive benchmarks show higher memory latencies. For 447.dealII (32242 cycles) and 450.soplex (22199 cycles), we identify that pathological bank access patterns yield more bank conflicts, thereby increasing the latency further.



(a) Performance.



(b) Update completion rate per epoch length for BSUP.



(c) Average update latency for BSUP (error bars denote min / max).

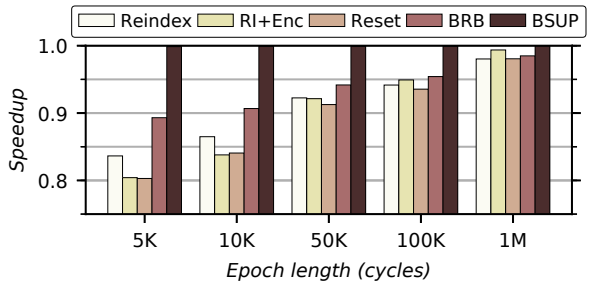
Fig. 11. BTB encryption results.

with 5K-cycle epoch, but it becomes 1.8% with 10K-cycle epoch. This demonstrates the impact of the length of the epoch.

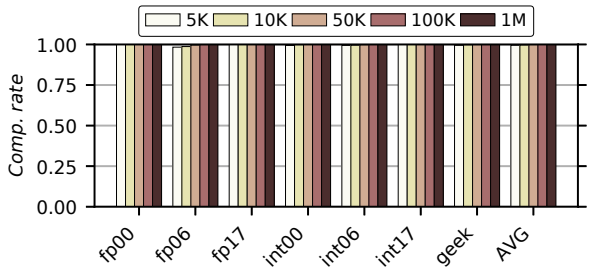
5.2 TAGE Results

TAGE has been one of the most accurate branch predictors for more than a decade. The latest improvement, TAGE-SC-L [36], has multiple components: a default bimodal predictor, TAGE predictors with different history lengths, a loop predictor, and a statistical corrector. The factors that make the TAGE predictor most accurate are 1) tagged entries, 2) geometric (longer) history lengths, 3) confidence metric, and 4) statistical corrector. We do not consider encryption for the loop predictor and the statistical corrector. Indexing TAGE predictors is very similar to the BTB, but different (longer) history lengths from the BHR are used for predictor tables. Thus, applying encryption to the set indexing is straightforward. On the other hand, data encryption is simpler

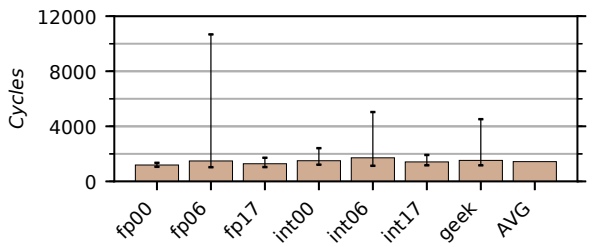
because we use a 3-bit saturating counter as opposed to a 47-bit target. We apply the two-level encryption to TAGE predictors while the BTB remain intact in this section.



(a) Performance.



(b) Update completion rate for BSUP.



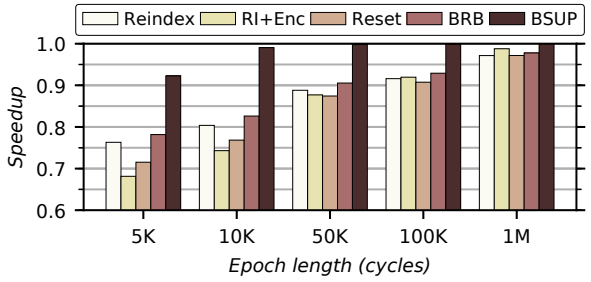
(c) Average update latency for BSUP.(Error bars denote min / max)

Fig. 12. TAGE encryption results.

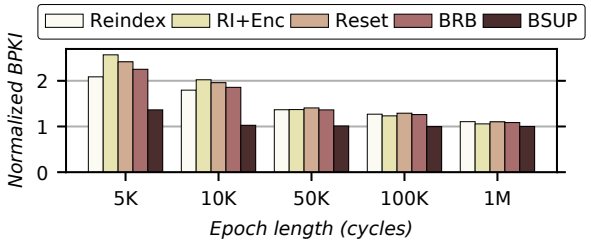
Figure 12a shows the performance impact of the TAGE predictor by different schemes, which shows a similar trend to the BTB results in Section 5.1. Reindex outperforms RI+Enc and Reset in shorter epoch lengths, but it has security risks. RI+Enc and Reset show very similar results. While the probability of an incorrectly decrypted BTB entry coincidentally leading to a correct prediction is extremely low, the direction prediction has a relatively higher chance of being correct even after using the wrong key. This makes RI+Enc perform slightly better. BRB performs much better than Reset by keeping the default bimodal predictor and the shortest history TAGE predictor, but it still shows 5 to 10% performance degradation in most epoch configurations. On the other hand, BSUP shows at most 0.1% performance degradation across all epoch lengths. The minimum latency to update all TAGE predictors is 1536 cycles (3 cycles x 512 sub-epochs), and the average update latency of

most benchmarks are under 5000 cycles, as shown in Figure 12c. Only two benchmarks have longer than 5000-cycle update latencies (447.dealII: 10676, 403.gcc: 5038). As a result, most benchmarks can complete the update process within a 5K-cycle epoch, as shown in Figure 12b.

5.3 Putting It All Together



(a) Performance



(b) Normalized branch misses per kilo-instruction (BPKI)

Fig. 13. Branch predictor encryption results.

We now evaluate the performance impact by the encryption on both BTB and TAGE predictors combined. Figure 13a shows the performance results. The overall trend is similar to individual predictor results, although the exact result is not the summation of both predictors because they operate in lock-step. Reindex still outperforms RI+Enc and Reset despite security concerns. The penalty of mispredictions in RI+Enc outweighs that of no-predictions in Reset for shorter epochs. BRB performs much better than Reset, but we still observe significant performance degradation compared to the baseline. With 5K-cycle epochs, we find 23.7%, 31.9%, 28.5%, and 21.8% for Reindex, RI+Enc, Reset, and BRB, respectively, while BSUP shows only 7.7% performance degradation. With 10K-cycle epochs, other schemes still show around 20% performance degradation, but BSUP shows only 1.0% degradation by quickly updating entries with a new encryption key. Figure 13b shows the normalized branch misses per kilo-instruction to the baseline (no encryption). We can confirm that the increased branch misses because of an incomplete set update lead to performance degradation.

BSUP increases the dynamic power consumption because of extra accesses to the predictors. Figure 14 shows the average access ratio of the BTB and the TAGE over the baseline. For the 4096-entry BTB, we need 4096 extra accesses per epoch. With a short epoch, additional accesses can be very significant (2.95x in 5K-cycle epoch). TAGE has smaller increases because of its relatively smaller size of the individual predictor table. The overhead by extra accesses becomes negligible at longer epochs.

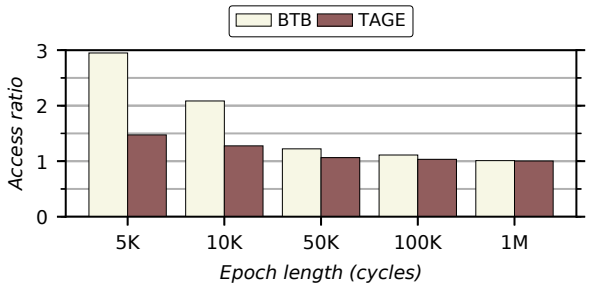


Fig. 14. BTB and TAGE access increases.

We summarize our findings as follows:

- The performance degradation by losing branch prediction information upon key changes is not trivial, in particular with shorter epochs.
- Reindex performs better than RI+Enc and Reset because of useful reuses despite security concerns.
- There is no clear answer of which is better: mispredictions in RI+Enc or no predictions in Reset. This can vary based on micro-architecture implementation. Other factors, such as implementation complexity and power consumption, should be considered.
- BRB performs better than Reset by keeping part of the prediction information.
- BRB on tagged structures (BTB) is less effective than tag-less structures (bimodal) because of the coverage.
- BSUP is effective even with very short epochs by updating prediction information quickly, but dynamic power consumption increases because of additional predictor accesses during an update.
- The hardware overhead of BRB is 5.88 KB (BTB: 1.88 KB, bimodal and TAGE predictors: 4 KB) while BSUP requires 654 B (BTB: 48B, bimodal: 2B, TAGE: 90B, others: 2B, key table: 512B).

5.4 BTB Size Sensitivity Test

The size of the branch predictor table directly affects the set update latency. A larger predictor can improve performance while the update takes longer. In this section, we perform a sensitivity study on the BTB size because it shows more sensitive results than the TAGE in terms of different epoch lengths. We vary the size of the L3 BTB from 1K to 16K entries (baseline: 4K). Figure 15 shows the results. Larger BTBs marginally improve performance (0.4% with 16K-entry) because the instruction footprint for most of the evaluated benchmarks fits well in 4K-entry. On the other hand, the update latency will be proportional to the size. For example, the minimum update latency is 1152 cycles with 1K-entry, but it requires 9216 and 18432 cycles for 8K and 16K entries, respectively. As a result, 8K and 16K-entry BTBs exhibit more than 10% performance degradation in 5K-cycle epoch and perform worse than smaller BTBs in 5K- and 10K-cycle epochs. This observation leads to an interesting design question: if a processor has very rigorous security requirements, and we want to implement an encryption mechanism for the branch predictor, we may consider a *smaller* branch predictor. Although this may increase the number of branch mispredictions in the steady-state, we can protect the branch predictor more aggressively.

From this experiment, we partially observe how partitioning affects performance. With one-half and one-quarter BTB entries, although the average across all suites shows little performance degradation (2.5% with 1K entries), some integer benchmarks show significant performance degradation. For example, 253.perlbmk shows 19.9% performance degradation with 1K-BTB. Based on how many

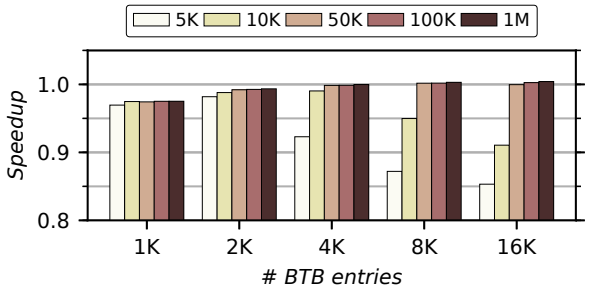


Fig. 15. BTB size sensitivity results.

concurrent contexts an OS supports, effective size can be further decreased, thereby losing more performance. Efficient dynamic partitioning can be useful, but it is beyond the scope of our work.

5.5 Bank-level Parallelism

We need to consider bank-level parallelism (BLP) to achieve quick set updates. Otherwise, the number of bank conflicts will increase, which in turn increases the update latency. In this section, we evaluate how BLP affects performance. Figure 16 shows the results. For BSUP (no BLP), we deliberately swap sets within the same bank. Consequently, the minimum update latency for the BTB increases from 4608 to 8704 (8 cycles to read, 1 cycle update, 8 cycles to write for 512 sub-epochs). We identify that the update completion rate for the BTB decreases from 9.2% to 0% and 92.2% to 17.8% for 5K- and 10K-cycle epochs, respectively. This results in 4.7% and 2.7% performance degradation. This confirms the benefit of a BLP-aware set update mechanism.

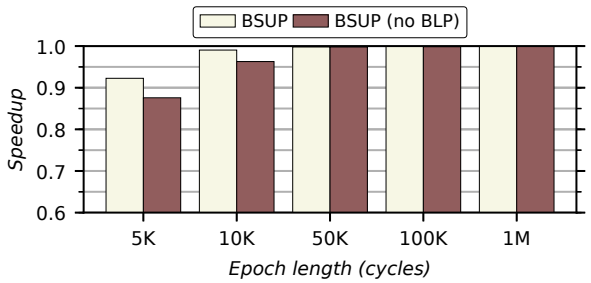


Fig. 16. BLP sensitivity results.

5.6 Instruction-based Epoch

We used cycle-based epochs so far because it can better demonstrate the relationship between the set update latency and the length of an epoch. However, cycle-counts may not well represent program behavior, such as instruction per cycle or branch intensity. In this section, we evaluate mechanisms with instruction-based epochs. Figure 17 shows the results. We vary the branch instruction count from 5K to 1M, which is chosen to be consistent with the cycle-based epoch. The overall trend does not differ much with previous results: 1) shorter epoch configurations incur

significant performance degradation, 2) BRB outperforms Reset, and 3) BSUP greatly outperforms any other schemes and performs well even with very short epochs.

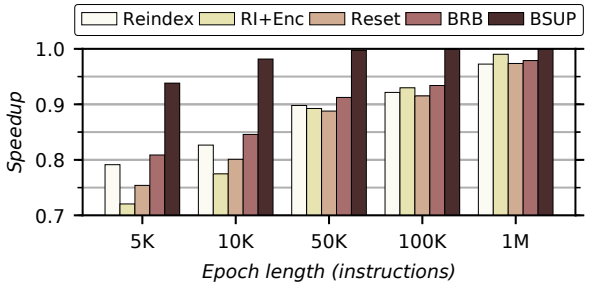


Fig. 17. Instruction-based epoch results.

6 RELATED WORK

6.1 Branch Predictor Side-Channel Attacks

Earlier branch predictor attacks target cryptographic algorithms [1–3]. By measuring timing differences caused by the penalty of a mispredicted branch in cryptographic primitives that use a data-dependent control flow, secret key information can be revealed. Evtyushkin et al. [8] exploited the BTB to bypass address space layout randomization, which adds random offsets in high-order bits of the virtual address of code and data segment, heap, and stack memory so that it is difficult for attackers to identify the exact address of victim code. However, BTB collisions still occur since the BTB is accessed by the low-order bits. Through timing side-channels, attackers identify which BTB entry is evicted by the victim's known branch instruction, then the random offset can be revealed.

The recent Spectre [20] variant 1 attack (array bounds check bypass) maliciously trains the branch direction in the predictor so that prediction leads to wrong-path instructions. On the other hand, the variant 2 attack (branch target injection) injects the target address of the malicious gadget code in the BTB. Then, a cache side-channel is used to extract secret information. Lee et al. [22] demonstrated a branch collision-based attack under the Intel SGX environment. Shadow branch instructions are deliberately designed to access the branch predictor entry updated by the victim application to infer fine-grained control flow information. BranchScope [9] extends collision-based attacks to infer the value of prediction counters using Prime+Probe, i.e., the adversary primes the prediction counters with the desired value, schedules the victim to run a branch instruction, and then tests the outcome of the shadow branch.

Return address stack (RAS) side-channels also exist. Upon function returns, the return address is predicted using the top of the RAS. In recent attacks [21, 24], an adversary can manipulate the value of the RAS so that a benign return instruction may speculatively redirect control flow to gadget code.

6.2 Branch Predictor Side-Channel Mitigations

6.2.1 Software-based Mitigations. Data-dependent branches can be algorithmically pruned. Also, conditional branches can be eliminated using conditional move instructions (cmov). However, these approaches are very algorithm-specific and may not apply to all applications. Also, if there are significant latency differences between branch instructions, this can be exploited using a timing side-channel attack. Another software mitigation is to obfuscate the control flow. For example,

ZigZagger [22] executes all branch instructions in if-else clauses. The branch target will be set using cmov instruction. At the end of a zig-zag pattern, it performs an indirect branch to the target. Also, control flow randomization [15] can be used such that jump blocks (or trampolines) will be executed regardless of the actual control flow by transforming conditional branches to unconditional indirect branches. To prevent reverse-engineering of trampolines, addresses of trampolines can be randomized. Retpoline [42] is proposed to mitigate Spectre variant 2 attacks, which is a software construct to isolate indirect branches from speculative execution. Speculative load hardening [6] implements a check for sensitive loads using branchless code, which is dependent on the opposite outcome of a conditional branch. Consequently, the address of load instructions in a wrong-path will be invalidated. These mechanisms proactively mitigate potential security breaches, so they can be effective with a higher performance overhead. OS-based time protection [11] can provide temporal isolation among processes by flushing on-core microarchitectures state on domain switch.

6.2.2 Tagged Branch Predictor. Branch predictors are tag-less structures in terms of contexts. TAGE predictors [36] use tags to improve prediction accuracy, but they cannot differentiate accesses from different contexts. To differentiate, we can add context IDs for each branch predictor entry. For example, Intel uses a 12-bit process-context ID (PCID), and Arm uses an 8-bit address space ID (ASID). In some cases, we need to differentiate the virtualization environment and privilege level.[9] Although adding identifiers will improve security, it incurs significant hardware overhead. For example, if we add a 20-bit identifier for a 1K-entry predictor, it requires a 2.5KB of storage.

Instead, we can reduce the overhead by keeping active contexts only. We maintain a small table that keeps full identifiers. Each branch predictor entry now keeps an index of the table rather than full identifiers. This can significantly reduce the storage overhead. However, false matching can occur because we reuse table indices.

6.2.3 Partitioning. Partitioning is a popular technique to provide isolation, but it reduces the effective size of a structure. It also has a scalability problem with a larger number of concurrent contexts. Without a smart dynamic partitioning mechanism, performance may be degraded significantly.

6.2.4 Randomization. An adversary can create branch collisions because the same index (hash) function is used for all contexts. If we use a different index function for each context, branch predictor accesses can be randomized. Even though an adversary manipulates the BHR and the low-order bits of the PC to match a target victim instruction, the resulting set ID will be different for each context.

6.2.5 Encryption. ScatterCache [44] uses an encryption mechanism to pseudo-randomly form a cache set with cache lines from different cache sets in the original cache construction. Also, this mapping changes based on security domains and their keys.

6.2.6 Checkpoint. Branch retention buffer (BRB) [43] is the state-of-the-art branch predictor side-channel mitigation. BRB maintains a small checkpoint (1.5 to 3KB) of branch predictor states for each context upon context switches and restores it once the context becomes active. Meanwhile, BRB flushes most components of the branch predictor. Checkpoint facilitates quick branch predictor warm-up. BRB can be considered as a partitioning mechanism since large structures are split into multiple partitions for each context. However, the storage overhead in checkpointing grows with more contexts. For faster switching, we cannot aggressively apply power-gating, so it increases leakage power. Otherwise, the corresponding table needs to be powered up upon context switches, which requires special power gating logic and additional latencies.

[9]Intel uses a 16-bit virtual processor ID (VPID) and four protection levels (Ring 0-3). Arm uses an 8- or 16-bit virtual machine ID (VMID) and also four exception levels (EL0-3).

6.2.7 Spectre Mitigation. Spectre attacks [20] set up the branch predictor to initiate the attack and use other side-channels such as TLB, page table, instruction and data caches, to transmit the secret information. Various mechanisms [19, 34, 35, 40, 46, 49] have been proposed to protect micro-architectural structures other than the branch predictor against Spectre attacks by delaying micro-architectural changes from speculative executions [19, 46, 49], recovering previous states, and randomizing accesses [34].

7 CONCLUSIONS

Side-channel attacks expose significant vulnerabilities in modern processors. In this paper, we proposed a two-level dynamic encryption mechanism for side-channel mitigation to protect modern branch predictors. The per-context encryption key is dynamically changed to overcome the weakness of static encryption. To alleviate performance degradation caused by the loss of prediction information upon key changes, we proposed a light-weight set update mechanism that also fully utilizes bank-level-parallelism in branch predictors. We show that our BLP-aware set update mechanism performs close to the insecure baseline even with very short epochs, 7.7% and 1.0% slowdown for 5K- and 10K-cycle epochs, respectively, while other mechanisms show significantly higher performance overhead.

ACKNOWLEDGMENTS

We thank Derek Miller, Chris Reed, Mathias Brossard, Prakash Ramrakhyani, Hanno Becker, Sangho Lee, and the anonymous reviewers for their suggestions and feedback on improving the paper.

REFERENCES

[1] Onur Aciiçmez, 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 (IMACC)* (Cirencester, UK). Springer-Verlag, Berlin, Heidelberg, 185–203.

[2] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2006. Predicting Secret Keys via Branch Prediction. In *Proceedings of the 2007 Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA)* (San Francisco, CA). Springer-Verlag, Berlin, Heidelberg, 225–242.

[3] 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 (ASIACCS)* (Singapore). ACM, New York, NY, USA, 312–320.

[4] Thomas Allan, Billy Bob Brumley, Katrina Falkner, Joop van de Pol, and Yuval Yarom. 2016. Amplifying Side Channels Through Performance Degradation. In *Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC)* (Los Angeles, California, USA). ACM, New York, NY, USA, 422–435.

[5] Arm. 2017. Arm TrustZone True Random Number Generator. https://static.docs.arm.com/100976/0000/trustzone_true_random_number_generator_technical_reference_manual_100976_0000_00_en.pdf.

[6] Chandler Carruth. 2018. Speculative Load Hardening. https://llvm.org/docs/SpeculativeLoadHardening.html.

[7] G. Cox, C. Dike, and D. J. Johnston. 2011. Intel's digital random number generator (DRNG). In *2011 IEEE Hot Chips 23 Symposium (HCS)*. IEEE Computer Society, Washington, DC, USA, 1–13.

[8] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: Attacking Branch Predictors to Bypass ASLR. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (Taipei, Taiwan). IEEE Press, Piscataway, NJ, USA, Article 40, 13 pages.

[9] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. 2018. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *Proceedings of the 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Williamsburg, VA, USA). ACM, New York, NY, USA, 693–707.

[10] Agner Fog. 2019. Instruction tables. https://www.agner.org/optimize/instruction_tables.pdf.

[11] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. 2019. Time Protection: The Missing OS Abstraction. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys)* (Dresden, Germany). ACM, New York, NY, USA, Article 1, 17 pages.

[12] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *Proceedings of the 27th USENIX Security Symposium (Security)*. USENIX Association, Baltimore, MD, 955–972.

[13] Thomas Christopher Grocutt and Yasuo Ishii. 2019. Encoding of input to branch prediction circuitry. https://patentscope.wipo.int/search/en/detail.jsf?docId=WO2019106333. US Patent 2019/0166158 A1.

[14] David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, Washington, DC, USA, 490–505.

[15] Shohreh Hosseinzadeh, Hans Liljestrand, Ville Leppänen, and Andrew Paverd. 2018. Mitigating Branch-Shadowing Attacks on Intel SGX Using Control Flow Randomization. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution (SysTEX)* (Toronto, Canada). ACM, New York, NY, USA, 42–47.

[16] Intel. 2020. Intel 64 and IA-32 Architectures Software Developer's Manual Volumn 3A. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf.

[17] D. A. Jimenez and C. Lin. 2001. Dynamic branch prediction with perceptrons. In *Proceedings of the 7th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, Washington, DC, USA, 197–206.

[18] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. 2016. A High-resolution Side-channel Attack on Last-level Cache. In *Proceedings of the 53rd Annual Design Automation Conference (DAC)* (Austin, Texas). ACM, New York, NY, USA, Article 72, 6 pages.

[19] 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 *Proceedings of the 56th Annual Design Automation Conference (DAC)* (Las Vegas, NV, USA). ACM, New York, NY, USA, 6.

[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 *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P)* (San Francisco, CA). IEEE Computer Society, Washington, DC, USA, 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)*. USENIX Association, Baltimore, MD.

[22] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *Proceedings of the 26th USENIX Security Symposium (Security)*. USENIX Association, Vancouver, BC, 557–574.

[23] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks Are Practical. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, Washington, DC, USA, 605–622.

[24] Giorgi Maisuradze and Christian Rossow. 2018. Ret2Spec: Speculative Execution Using Return Stack Buffers. In *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security (CCS)* (Toronto, Canada). ACM, New York, NY, USA, 2109–2122.

[25] Scott Mcfarling. 1993. Combining Branch Predictors. In *Technical Report TN-36, Digital Western Research Laboratory*.

[26] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Proceedings of the 2006 Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA)* (San Jose, CA). Springer-Verlag, Berlin, Heidelberg, 1–20.

[27] Arthur Perais, Rami Sheikh, Luke Yen, Michael McIlvaine, and Robert D. Clancy. 2019. Elastic Instruction Fetching. In *Proceedings of the 25th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, Washington, D.C., 478–490.

[28] Colin Percival. 2005. Cache missing for fun and profit. In *Proceedings of the Technical BSD Conference (BSDCan 2005)*.

[29] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. 2003. Using SimPoint for Accurate and Efficient Simulation. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)* (San Diego, CA, USA). ACM, New York, NY, USA, 318–319. https://cseweb.ucsd.edu/~calder/simpoint/.

[30] Primate Labs. 2019. Geekbench. https://www.geekbench.com/.

[31] Moinuddin K. Qureshi. 2018. CEASER: Mitigating Eviction-Based Cache Attacks via Dynamically Encrypted Address. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (Fukuoka, Japan). IEEE Computer Society, Washington, DC, USA, 775–787.

[32] Moinuddin K. Qureshi. 2019. New Attacks and Defense for Encrypted-address Cache. In *Proceedings of the 46th Annual International Symposium on Computer Architecture (ISCA)* (Phoenix, Arizona). ACM, New York, NY, USA, 360–371.

[33] Charles Reis, Alexander Moshchuk, and Nasko Oskov. 2019. Site Isolation: Process Separation for Web Sites within the Browser. In *Proceedings of the 28th USENIX Security Symposium (Security)*. USENIX Association, Santa Clara, CA, 1661–1678.

[34] 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 (MICRO)* (Columbus, OH, USA). ACM, New York, NY, USA, 73–86.

[35] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. 2019. Efficient Invisible Speculative Execution Through Selective Delay and Value Prediction. In *Proceedings of the 46th Annual International Symposium on Computer Architecture (ISCA)* (Phoenix, Arizona). ACM, New York, NY, USA, 723–735.

[36] Andre Seznec. 2014. TAGE-SC-L Branch Predictors again. 5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5).

[37] André Seznec, Stephen Felix, Venkata Krishnan, and Yiannakis Sazeides. 2002. Design Tradeoffs for the Alpha EV8 Conditional Branch Predictor. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)* (Anchorage, Alaska). IEEE Computer Society, Washington, DC, USA, 295–306.

[38] Youngjoo Shin, Hyung Chan Kim, Dokeun Kwon, Ji Hoon Jeong, and Junbeom Hur. 2018. Unveiling Hardware-based Data Prefetcher, a Hidden Source of Information Leakage. In *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security (CCS)* (Toronto, Canada). ACM, New York, NY, USA, 131–145.

[39] Standard Performance Evaluation Corporation. 2017. SPEC CPU Benchmark Suites. https://www.spec.org/cpu/.

[40] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. 2019. Context-Sensitive Fencing: Securing Speculative Execution via Microcode Customization. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Providence, RI, USA). ACM, New York, NY, USA, 395–410.

[41] Eran Tromer, Dag Arne Osvik, and Adi Shamir. 2010. Efficient Cache Attacks on AES, and Countermeasures. *Journal of Cryptology* 23, 1 (Jan. 2010), 37–71.

[42] Paul Turner. 2018. Retpoline: a software construct for preventing branch-target-injection. https://support.google.com/faqs/answer/7625886.

[43] Ilias Vougioukas, Nikos Nikoleris, Andreas Sandberg, Stephan Diestelhorst, Bashir M. Al-Hashimi, and Geoff V. Merrett. 2019. BRB: Mitigating Branch Predictor Side-Channels. In *Proceedings of the 25th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, Washington, D.C., 466–477.

[44] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. 2019. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In *Proceedings of the 28th USENIX Security Symposium (Security)*. USENIX Association, Santa Clara, CA, 675–692.

[45] Y. Xu, W. Cui, and M. Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, Washington, DC, USA, 640–656.

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

[47] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proceedings of the 23rd USENIX Security Symposium (Security)*. USENIX Association, San Diego, CA, 719–732.

[48] Tse-Yu Yeh and Yale N. Patt. 1991. Two-level Adaptive Training Branch Prediction. In *Proceedings of the 24th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (Albuquerque, New Mexico, Puerto Rico). ACM, New York, NY, USA, 51–61.

[49] 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 (MICRO)* (Columbus, OH, USA). ACM, New York, NY, USA, 954–968.

[50] Tao Zhang, Kenneth Koltermann, and Dmitry Evtyushkin. 2020. Exploring Branch Predictors for Constructing Transient Execution Trojans. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Lausanne, Switzerland). ACM, New York, NY, USA, 667–682.