BeKnight: Guarding against Information Leakage in Speculatively Updated Branch Predictors

Md Hafizul Islam Chowdhuryy
University of Central Florida
Orlando, FL, USA
reyad@knights.ucf.edu

Zhenkai Zhang
Clemson University
Clemson, SC, USA
zhenkai@clemson.edu

Fan Yao
University of Central Florida
Orlando, FL, USA
fan.yao@ucf.edu

***Abstract*—Information leakage through processor microarchitectural components exploiting speculative execution is raising significant security concerns. Modern commercial processors incorporate branch predictor designs where internal states of branch predictor structures are speculatively updated. Recent studies have shown that speculatively updated branch predictors allow side channel exploitation in the speculative domain, extending branch predictors to be another source of transmitting medium in transient execution attacks. While postponing updates of branch predictor states at a later time (e.g., during commit) can avoid exploitation in the speculation domain, it can result in belated correction of prediction outcomes (e.g., branch direction), leading to non-trivial degradation of prediction performance.**

In this paper, we present BeKnight, a secure branch predictor design that defeats *speculative* side channels targeting the branch direction prediction structure as the source of leakage. BeKnight aims to *retain the performance advantage of early branch predictor updates* (i.e., at resolution time) while ensuring no transient leakage. To achieve this, BeKnight conscientiously tracks the ownership and speculative changes of potentially unsafe pattern history entries using a small Speculative Pattern Lookaside Buffer (SPLB). BeKnight efficiently audits the use of pattern history by allowing subsequent predictions in the same domain to benefit from early updates while annulling potential leakage through ensuring *architecturally correct* pattern is used on detection of a domain conflict. We evaluate the performance of BeKnight using 24 representative workloads from SPEC-2017. Notably, BeKnight achieves almost identical performance compared to the system with insecure but performant speculatively-updated predictors.

I. Introduction

Modern processors heavily rely on speculation to offer high instruction level parallelism. Particularly, branch prediction unit (BPU) performs branch direction and target predictions, enabling the processor to continuously feed instructions to the front-end to minimize pipeline stalls. Branch mispredictions introduce wrong-path execution of instructions, also known as transient execution. As transient execution defies the expected program semantic, the underlying speculation hardware is designed to ensure no architectural state changes are remained due to wrong-path execution. However, *microarchitectural states* (e.g., in caches) may be altered but not properly purged, leading to transient execution attacks [1].

Compared to classical side channels [2]–[8], transient execution attacks can exfiltrate program unintended secrets (e.g., cross boundary or security domain) [1], [9]–[12]. Typically, such attacks use branch predictors as the *triggering* mechanism to induce wrong-path executions. Recent works [10], [13] have revealed modern processors (e.g., Intel CPUs)

```
1  secret[] = {0,...,1};
2  data[SIZE];
3  if (idx < SIZE)     //b_p: trigger speculation
4    x = data[idx];    //speculative load
5    if (x) y++;       //b_v: nested speculation
6    else y--;
```

Listing 1: Transient execution attacks in speculatively-updated branch predictors. data[idx] points to **secret** in speculation.

integrate *speculatively-updated branch predictors*. In particular, the state of branch prediction structures, such as the pattern history table (PHT), is updated at the resolution of conditional branches in the speculation path (e.g., in nested speculation). Moreover, these alterations are not cleared after speculative branches are eventually squashed. This makes branch predictors vulnerable to information leakage in the speculation domain, rendering BPUs exploitable as the *transmitting medium* in transient execution. Listing 1 shows a code snippet illustrating this vulnerability. Victim has two branches, b_p and b_v, where branch b_v can execute transiently under the shadow of b_p. Attacker controls idx to point to the unintended sensitive values (**secret**) and trains the PHT to make b_p mispredict. If b_v gets resolved before b_p's misprediction is detected, the BPU will update the states of PHT based on b_v's predicate. Later when b_v is squashed, the state change made by transient execution of b_v remains. The attacker then executes a branch congruent to b_v and measures its timing to infer the value of **secret** (see Section III for details).

Speculation-based BPU side channels are extremely dangerous since 1) they can potentially target *any branch* in nested speculation, thus defensive mechanisms avoiding direct control/data flow dependency on secrets are insecure [13]. For instance, in Listing 1, **secret** is not directly operated on using any branches; 2) such attacks leverage the branch predictor exclusively for both transient execution trigger and speculative secret transmission. Therefore, they can manifest even when all other microarchitectural components are not exploitable. In particular, the code in lines 5 and 6 of Listing 1 can not be exploited through other mediums such as cache since the instructions are in the same cache line and the same data (i.e., variable y) is accessed in each branch direction. Recently, several mitigation techniques have been proposed to improve BPU microarchitecture security [11]. Specifically, resource isolation [14], [15] and access randomization techniques [15]–[17] mitigate leakage by disabling modulating or inferring microarchitectural states. While such schemes can potentially avoid using BPU as a classical side channel, they typically suffer from low scalability, non-trivial performance overhead, and hardware cost. Secure oblivious speculation frameworks (e.g., [18]–[22]) prevent speculative secrets propagation by delaying microarchitectural state changes until the corresponding instructions are deemed to be committed. While these approaches can defeat a wide spectrum of transient execution attacks, they do not evaluate the positive performance impact of early speculative state updates of BPU.

In this work, we propose ***BeKnight***, a low-cost secure speculation design for speculatively-updated branch predictors that guards against side channels in the speculation domain. Motivated by the sensitivity of time-to-update for BPUs, BeKnight aims to retain the performance advantage for speculatively-updated branch predictors while ensuring that speculation-domain leakage is disbanded. Particularly, BeKnight dynamically tracks the ownership of *unsafe PHT entries* exploitable in the speculation domain. To provide both post-speculation security and in-speculation security, BeKnight maintains a small *speculative pattern lookaside buffer* (**SPLB**) that records the speculative updates of SPLB entries while maintaining the architecturally correct state in-place. BeKnight *opportunistically* uses the *unsafe* speculative updates from SPLB to benefit prediction for same-domain accesses while ensuring cross-domain accesses are served through leakage free PHT. We implement a prototype of BeKnight and evaluate its efficacy with 24 single-program and multi-program workloads built from SPEC-2017. Our evaluation shows BeKnight offers security guarantee against BPU attacks in the speculation domain while exhibiting almost similar performance as insecure but performant BPU in modern processors with negligible hardware cost. In summary, the contributions of our work are:

- We explore the impact of speculative updates of BPU and systematically investigate the rationale of speculatively-updated branch predictors in commercial processors.
- We perform extensive characterization of PHT updates due to branches in transient execution path among various workloads, motivating the design of low-overhead tracking mechanisms for unsafe speculative updates.
- We propose BeKnight, a secure branch predictor that defeats the use of direction prediction as a *transmitting medium* in transient execution attacks. BeKnight is fully integrated with modern branch predictor that can eliminate any speculation leakage through on-demand clearing of *non-architectural* states of PHT.
- We implement BeKnight and evaluate its efficacy using single- and multi-program workloads. Results shows BeKnight can effectively thwart BPU speculation leakage with very low performance overhead.

II. Background

Branch Prediction Unit. BPU is a per-core structure that dictates the control flow of program execution. BPU at a high level presents two primary functions: direction prediction (e.g., *taken* or *not taken*) and target address predictions (e.g., for indirect jumps). BPU generally maintains both pattern

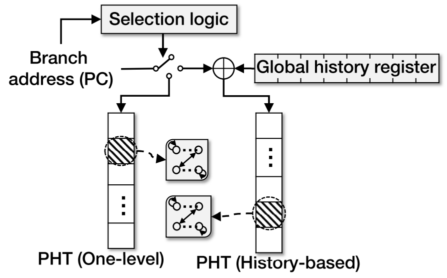


Fig. 1: Direction prediction mechanism in modern BPU.

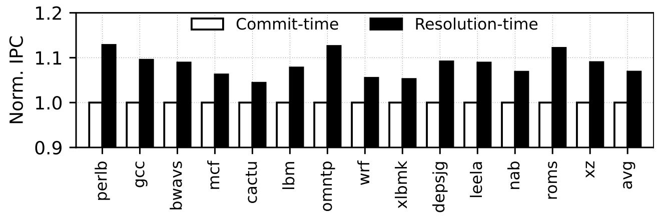
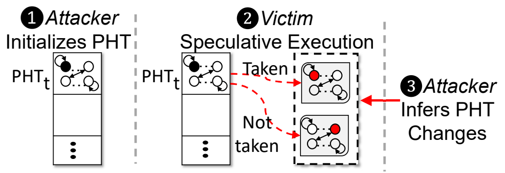


Fig. 2: Performance of *commit-time* and *resolution-time* update of PHT without restoration for misprediction.

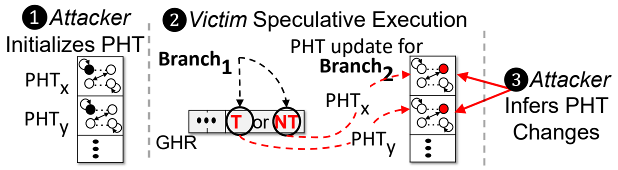
history (i.e., in PHT) and branch history (i.e., in Global History Register or GHR) to aid *direction predictions*. In *one-level prediction* mode, the predictor merely uses the branch instruction address to index into the PHT. Differently, the *history-based prediction* mode leverages global branch history in GHR along with branch address for PHT access (e.g., gShare [23]). BPU often harnesses a hybrid scheme where a selection logic is integrated to dynamically choose the predictions from the best performing branch predictors [2]. Figure 1 illustrates a hybrid branch predictor similar to the designs in recent Intel processors [2].

Microarchitectural Timing Channel Attacks. Side channels in hardware have been demonstrated on various processor components, such as TLBs, caches and branch predictors [2], [3], [6], [24]–[26]. Transient execution attacks [1], [11], [12], [27] enable an adversary to access unintended data in the system speculatively and subsequently extract them via a microarchitectural side channel. While non-speculative side channels rely solely on the victim's secret-dependent control flow or data flow [2], [9], [13], [28], transient execution attacks are even more perilous as they significantly extend the attacker's data access.

Attacks on Branch Predictors. Existing attacks on BPUs in the non-speculative domain exploit secret-dependent control flow, where a specific PHT entry is updated based on program-defined secrets like cryptographic keys. By observing the prediction behavior of a related branch using timing, attackers can recover the secret. Previous works have demonstrated both side and covert channels through such exploits [2], [3], [7]. PHT collisions have been shown in both one-level predictors [2] and history-based predictors [13]. It is worth noting that incorporating secret-dependent control flow is known to introduce side channels. These attacks can be mitigated by avoiding explicit conditional branching on secretive program



(a) Potential leakage of PHT



(b) Potential leakage of GHR

Fig. 3: Leakage from speculatively updated direction predictor.

data, which is commonly practiced in many cipher implementations such as OpenSSL DSA/ECDSA and GnuPG [29]. Recent studies [13], [27] reveal that BPUs in commercial processors update the PHT based on branch resolutions, even if those branch instructions are later discarded. Speculative updates of branch predictors significantly expand the attack surface to the speculative domain. These BPU attacks are particularly dangerous because any conditional branch in the transient execution path can leak information.

III. Motivation and Problem Analysis

Modern processors employ deep pipeline stages capable of handling long speculation paths, with speculation window extending up to thousands of cycles [18], [27]. To prevent pipeline stalls, processors employ nested speculation, executing subsequent branches while the earlier branch remains unresolved. Consequently, there is a possibility that later branches are resolved before the legitimacy of their execution path is verified. The performance of branch prediction can be significantly influenced by the timing of branch predictor updates for *speculatively* executed branches [30]–[32]. Our investigation reveals that commercial processors, including recent Intel processors, update the PHT at *branch resolution time*. More importantly, transient branch executions do not restore PHT states after speculation is corrected. We hypothesize the major reason is that reducing the number of cycles between branch prediction and branch predictor update can improve branch prediction performance in deeply pipelined processors [30]. Specifically, updating the PHT at resolution time allows training of the corresponding PHT entry *in advance*, enabling correct predictions for subsequent branch instructions fetched before instruction commit. Even in cases where branch resolution occurs in the wrong execution path, speculative PHT updates can be beneficial since the branch conditions may match architecturally correct data.

To understand the design rationale of early pattern history updates, we model a modern history-based branch predictor (See Section VIII for more details). We evaluate two PHT update policies: commit-time and resolution-time update. Figure 2 illustrates the performance comparison between these two configurations. Note that in both cases, the GHR is updated on *prediction* and restored for incorrect predictions during resolution [18], [33]. As shown, *resolution-time* update exhibits higher performance, surpassing the *commit-time* update by an average of 8.5%. Achieving superior performance in an already well-optimized microarchitectural component

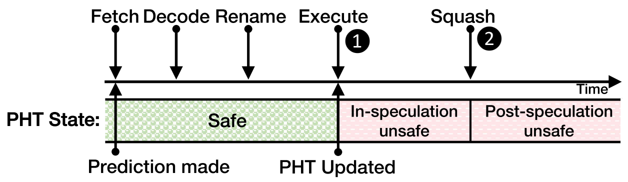


Fig. 4: Evolution of PHT entry from *safe* to IS-Unsafe and finally PS-Unsafe for different pipeline stages of a branch.

like the branch predictor poses significant challenges [32], [34]. Our study shows that speculative update of PHT brings advantageous in terms of performance, which likely explains why modern processors employ such a scheme [27].

Security Implications of Speculatively-updated BPU. Unfortunately, speculative updates of PHT introduce the PHT counter encoding vulnerability (i.e., infer if the victim branch is speculatively taken or not taken) [13], [27]. Specifically, the attacker can observe ⓐ *the direction of a specific victim branch*, whose outcome depends on a speculative secret (example shown in Listing 1); and ⓑ *the execution of a branch in a specific path* based on certain condition of a speculative secret. As illustrated in Figure 3, attacks on these primitives can be fundamentally classified as:

i) Leakage from PHT (Figure 3a): Leakage from PHT can manifest in three steps [13], [27]: ❶ preset the PHT entry (PHT_t) to a pre-determined state; ❷ trigger victim's execution where a branch resolution updates PHT_t according to a secret value in the speculative domain; ❸ infer either specific state change of PHT_t (ⓐ) or if PHT_t has been updated in the speculative path (ⓑ) by executing a congruent branch and timing its execution. Depending on execution latency of congruent branch with a known outcome, PHT_t is inferred.

ii) Leakage of GHR (Figure 3b): As GHR records the most recent branch outcomes, leakage of GHR can lead to recovery of the victim's control flow [13]. Although the state of the GHR cannot be directly observed, it determines which specific PHT entry is used for prediction. The leakage of GHR follows these steps: ❶ preset potential PHT entries (PHT_x and PHT_y) and the GHR state; ❷ trigger victim execution, where the update of GHR (and PHT) for $branch_1$ depends on a speculative secret; ❸ infer which PHT entry (PHT_x or PHT_y) is changed by the subsequent branch ($branch_2$), thereby leaking the updated state of GHR caused by $branch_1$.

In this paper, we aim to design secure speculatively-updated BPU that offer *same level of security as commit-time update*

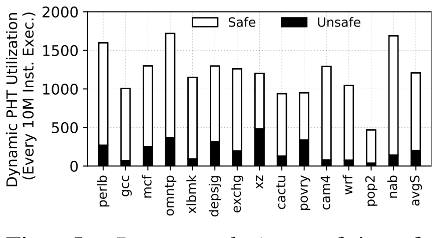


Fig. 5: *Post-speculation* safe/unsafe PHT entries in a history-based predictor. *Safe* entries are speculative leakage free, *Unsafe* entries are vulnerable.

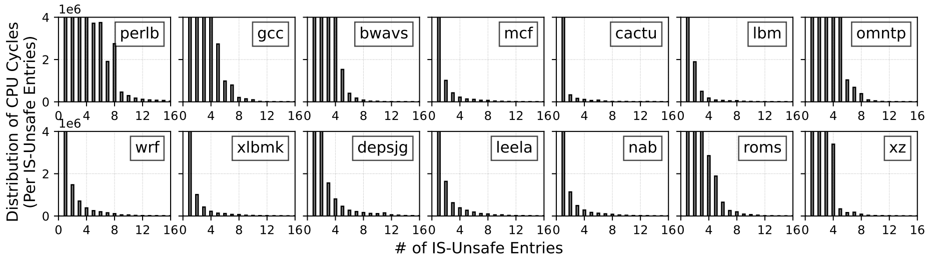


Fig. 6: Runtime distribution of PHT entries updated by in-flight branch instructions (i.e., potentially IS-Unsafe) during every CPU cycle of simulation, collected for a contiguous period of 100M cycles.

predictor while maintaining performance advantage.

IV. Threat Model

Our threat model focuses on the direction predictor-based BPU side channels in the speculation domain (i.e., [13], [27]). We assume an *active attacker* (similar to [35]) who can manipulate the branch predictor state and interfere with the victim's execution. The attacker has the ability to initiate victim execution and can be co-located on the same physical core as the victim process. The attacker and victim processes can run either in simultaneous multithreading (SMT) mode or in a round-robin (RR) fashion. We assume that the branch predictor speculatively updates PHT entries without restoration. By exploiting the vulnerability described in Section III, the attacker can launch transient execution attacks via BPU as a transmitting medium [27]. Our approach focuses on cross-domain attack scenarios and does not consider same-thread attacks, such as malicious trojans injected into benign victim applications [12]. Preventing same-thread attacks typically involves careful validation of application code or relies solely on trusted vendor-provided linked libraries. Protection against speculative and non-speculative side channels originating from other hardware components (e.g., cache and TLB [6], [24], [28]) falls outside the scope of this work.

V. Characterizing PHT Safe and Unsafe Usage

In speculatively-updated BPUs, PHT updates due to resolution of a conditional branch in the wrong path of execution can propagate secrets in speculative domain. To quantify how many PHT entries might potentially be perturbed at a certain time of program execution, we run characterization experiments on a set of benign programs and track speculatively-updated PHT entries. We categorize these PHT entries into two classes: i) ***Safe Entries:*** entries that have only been updated by branches that are later committed (i.e., these entries have not been impacted by any squashed instruction) and ii) ***Unsafe Entries:*** entries updated by *at least* one squashed branch. We further differentiate the *Unsafe* entries as *In-speculation unsafe* (**IS-Unsafe**) and *Post-speculation unsafe* (**PS-Unsafe**). The IS-Unsafe PHT entries are the ones that are updated by in-flight resolved instructions (not yet squashed), while PHT entries updated by branch instructions that are already squashed are the PS-Unsafe entries. Figure 4 illustrates different states of PHT entry, highlighting the difference between the two types of unsafe PHT entries. When the branch is executed (resolved), it updates a PHT entry and changes the PHT entry from safe to IS-Unsafe (❶). Once the branch instruction is squashed, the PHT entry transitions from IS-Unsafe to PS-Unsafe (❷). An attacker can recover secrets while the PHT entry is in IS-Unsafe state only in an SMT-enabled system. In contrast, an attacker can recover secrets from PS-Unsafe PHT entry in both SMT and non-SMT settings. We run representative regions of several SPEC 2017 CPU benchmarks and log their PHT entry update traces for continuous execution of 100 program segments (each with 10M instructions). Figure 5 shows the breakdown of PHT entries exercised and marked as safe or unsafe (averaged over all segments for each program). We observe that *only a very small number of PHT entries are tagged as unsafe* (on average 202) and can be a potential source of leakage in the speculative domain. Note that PHT typically has tens of thousands of entries (e.g., 16K in Intel processors [2]). Such observation confirms that branch predictors are generally highly accurate after training period. As a result, only a very small percentage of conditional branches resolved in the wrong path of execution will alter PHT states.

To understand IS-Unsafe, we count the total number of PHT entries corresponding to branch instructions executed (but not squashed or not in the head of ROB) cycle by cycle. At resolution time, since the processor can not determine if a branch instruction will be committed, strictly identifying IS-Unsafe entries are not possible. Hence we consider all in-flight resolved branch instructions as *potentially* IS-Unsafe. Figure 6 shows the runtime distribution of potentially IS-Unsafe entries. For most of the cycles, fewer than eight PHT entries are considered potentially IS-Unsafe. This observation indicates that for almost the entire speculation window, a small number PHT entries are being updated by in-flight branches. Note that while disabling SMT can prevent IS-Unsafe leakage, such a mechanism cannot prevent PS-Unsafe based leakage across processes running in the same physical core.

VI. BeKnight Design Framework

In this section, we present our BeKnight design that prevents both IS-Unsafe and PS-Unsafe leakage in branch predictors.

A. Securing Leakage via IS-Unsafe and PS-Unsafe Entries

To thwart all types of BPU-based transient execution attacks, the underlying security mechanism of BeKnight book-

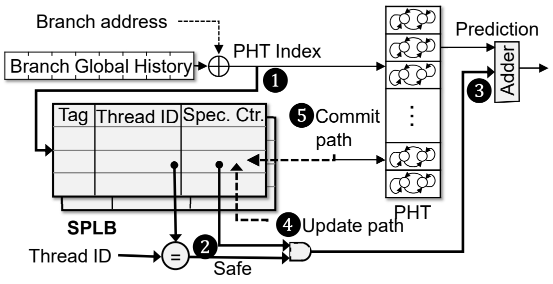


Fig. 7: Overview of BeKnight framework. Shaded areas represent added components; bold paths denote modified/added paths during prediction.

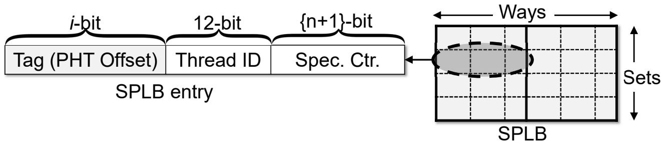


Fig. 8: Organization of SPLB. For n-bit PHT, *Speculative counter* (S) is {n+1}-bit. *Tag* field depends on the actual size of PHT and # of sets in SPLB (i.e., if # of PHT entries is P and # of sets is S, then # of PHT offset bits $i = \frac{P}{S}$).

keeps the ownership of unsafe PHT entries (both IS-Unsafe and PS-Unsafe) at any point of execution. To ensure security, the unsafe PHT entries that belong to one thread context cannot be used to make predictions on a different context using the *non-architectural states* (i.e., changes to PHT by instructions that are either transient or not committed yet), while the same domain prediction is allowed to retain the performance gain due to speculative updates. However, extending the PHT with metadata to track unsafe entries can be overly expensive. For example, considering a PHT with 2-bit counters, storing a 12-bit thread id along with a single-bit flag (to indicate if unsafe) increases the storage of PHT by $85\times$. This high-overhead mechanism still can not *rollback* to the architecturally correct state of PHT. Our secure BPU design is based on the observations (Section V) that protecting only unsafe PHT entries can annul speculative leakage, and more importantly the number of such entries are deemed small and can be efficiently tracked in a small buffer.

Figure 7 illustrates a high-level overview of BeKnight. At the core of our design is the integration of a small-size buffer that is utilized to track and audit the usage of unsafe PHT entries. We term such buffer structure as *speculative pattern lookaside buffer* (SPLB). Essentially, SPLB manifests similarly as translation lookaside buffer (TLB) that tracks the ownership of pages to processes. During the prediction of a branch instruction, BeKnight leverages SPLB to detect cross-domain accesses to unsafe entries. In case such access occurs, the proposed mechanism ensures that the prediction is provided based on the *architecturally correct* state, thus preventing prediction decisions made to be influenced by speculatively accessed data. Different from the speculative PHT update mechanism in modern processors, BeKnight works by keeping the PHT updated using only committed instructions, ensuring that an architecturally correct state is always maintained in PHT. BeKnight tracks the speculative PHT updates separately in SPLB and uses that for prediction only when same-domain access is detected. BeKnight exerts the fact that cross-domain accesses to the same PHT entry are effectively close to randomization of that entry since the actual behavior of branches across different domains are typically not correlated with each other. And hence, *reading* an architecturally correct counter ensures that there is no *speculative interference* on the PHT states.

B. BeKnight Design Details

The purpose of SPLB is to track the unsafe PHT entries and their ownership (e.g., the thread context that has transiently updated the PHT). To achieve that goal, each buffer entry incorporates three fields: 1) PHT offset (`pht_off`) that it tracks, 2) thread ID (`tid`) for the instruction allocating the entry, and 3) {n+1}-bit signed *Speculative counter* (S) indicating the *in-flight* and *transient* updates to this PHT entry. n is the number of bits in PHT counter. An {n+1}-bit signed counter is large enough to hold the necessary *transient updates* to the PHT entry since an overflow in S will saturate the actual PHT counter. This buffer is arranged in sets and ways (similar to TLBs) to facilitate lookup and update. Figure 8 illustrates detailed structure of SPLB. Specifically, SPLB works in parallel with the rest of the BPU. Upon making a direction prediction, a lookup in SPLB is performed by supplying `pht_off` and `tid` of the requesting domain. An SPLB lookup can result in one of three possible outcomes: i) buffer hit: where both `pht_off` and `tid` match with an existing entry in the buffer, meaning the S of this PHT is safe, ii) domain conflict: the `pht_off` matches with an existing entry but `tid` mismatches (i.e., a different thread context attempts to probe this entry), meaning the S of this PHT is unsafe, and only the architectural state maintained in PHT must be used, and iii) buffer miss: `pht_off` does not match with any existing entry. In case of buffer miss or domain conflict, the corresponding PHT entry is used for prediction since S is either not present or unsafe. However, in the case of a buffer hit, S can be safely used to make prediction.

Interactions of BeKnight with other BPU components. Now we describe the detailed operations and interactions of BeKnight in different stages of pipeline. BeKnight has explicit interactions in several different pipeline stages: 1) ensuring prediction using safe PHT entries during the *fetch* stage, 2) facilitating speculative update of the PHT and tracking IS-Unsafe at the end of *execute* stage and 3) marking safe/PS-Unsafe during the *commit* stage.

Fetch stage: At the fetch stage, by default, the BPU provides a prediction of the direction for a conditional branch by accessing the corresponding PHT entry. Since the PHT in BeKnight is only updated by committed instructions, it is safe from any speculative leakage. BeKnight additionally performs an SPLB lookup to determine the status of this specific PHT entry in parallel (❶). If the lookup returns buffer miss or

Processor	4-core, SMT, Out-of-order, x86, 3.0GHz
Pipeline	14-stage, 8-issue, 192-entry ROB, 32-entry Load/Store Queue
Cache	64KB 4-way L1 I-/D-cache (Private), 16MB 16-way L2 cache (Shared)
Memory	8GB, DDR3-based 1600MHz
Branch predictor unit	16K PHT, 3-bit saturating counter, 64-bit GHR

TABLE I: Architecture configuration parameters.

Multi-program Workload	**mix-1**: xz-deepsjeng, **mix-2**: mcf-gcc, **mix-3**: xlbmk-perlbench, **mix-4:** omnetpp-gcc, **mix-5**: mcf-xlbmk, **mix-6:** xz-omnetpp, **mix-7**: gcc-xlbmk, **mix-8:** omnetpp-xlbmk, **mix-9:** perlbench-deepsjeng, **mix-10:** xlbmk-xz

TABLE II: List of multi-programmed workload.

domain conflict, only the actual PHT counter is used to make prediction (❷). However, in case of SPLB hit (which represents same-domain access to an unsafe counter), BeKnight combines the corresponding `S` with the actual PHT counter (❸) to provide prediction based on speculative updates in PHT.
Execute stage: At the end of the execute stage, the branch outcome is resolved and the BPU performs a speculative update of the PHT based on the actual direction (regardless of whether it will be committed or squashed). Instead of directly updating the PHT (❹), i) for SPLB miss, an SPLB entry is allocated and `S` is updated based on the resolved outcome; ii) for domain conflict, the `S` is cleared (reset to 0) first to discard any speculative state and then both `tid` and `S` is updated; and finally iii) for SPLB hit, the `S` is updated based on branch resolution to keep track of speculative update.
Commit stage: At the commit stage (❺), the PHT counter is updated to maintain architecturally correct PHT state for correct path execution. In addition, in case of SPLB hit, `S` also has to be updated *in the opposite direction of branch direction* to maintain only the speculative updates in SPLB. For SPLB miss or domain conflict, this does not have to be done since speculative states of this PHT are not tracked.

If SPLB allocation during *execute stage* requires eviction of an SPLB entry, then one entry from the corresponding SPLB set is randomly selected for eviction and the `S` is discarded for that entry since it is no longer tracked. BeKnight uses random replacement for the SPLB so that no information can be inferred from the replacement policy. Moreover, since only architecturally updated PHT states are visible from a different domain, an attacker cannot determine if a PHT entry is tracked by SPLB or not. Finally, SPLB operations (e.g., lookup, update, eviction) are completed within the same cycle as the regular BPU access, hence it does not introduce additional variant timing sources. Our scheme leverages the idea that PHT itself will contain only architecturally correct states and the SPLB tracks the speculative updates, allowing us to benefit from them for same-domain access.

VII. Experimental setup

Architecture Configuration. We implement a prototype of the BeKnight framework in gem5 [36] with full system simulations. We boot a Ubuntu 18.04 based system using kernel version 4.19.83. We model an X86-based processor with a pipeline configuration close to Intel Skylake [37]. We model a history-based branch predictor using the details revealed in the BPU design of recent Intel processors [2], [13], [27]. Table I lists the detailed architecture configuration. As default configuration, we use a 4-way associative 128-entry SPLB.

Workload Configurations. We build a representative set of workloads including 14 single-program and 10 multi-program workloads from the SPEC CPU 2017 benchmark suite with reference inputs. Table II details the workload configurations. For the single-program workloads, we skip the first 6 billion instructions after program execution, then perform 1 billion instructions simulation in out-of-order simulation. For the multi-program workloads, we use both *round-robin (RR)* and *SMT* configuration. In the RR configuration, we simulate 1 billion instructions in the region of interest for both benchmarks combined. For SMT configuration, we simulate at least 1 billion instructions for each of the workloads individually.

Evaluation Methodology. To evaluate BeKnight, we configure two baseline schemes with different security implications, including i) *Commit-time update, **CT**:* secure baseline scheme which performs PHT updates at only commit time, ii) *Resolution-time update without restore, **RT-NR**:* this is employed in commercial processors where PHT update is performed at branch resolution time without restoration for squashed branch executions. Note that this configuration is vulnerable to exploitation discussed in Section III. Finally, both configurations have speculative update of GHR.

	Non-SMT		SMT	
	Direction	Execution	Direction	Execution
PHT Leakage	✓	✓	✓	✓
GHR Leakage	✓	✓	✓	✓

TABLE III: BeKnight protection in spec.-updated BPU.

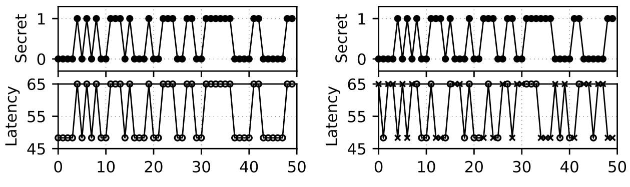


Fig. 9: Execution latency of b_a (cycles) corresponding to actual direction (based on speculatively accessed secret) of b_v executed in wrong path of speculation. The circled and crossed points denote correct and incorrect secret guesses.

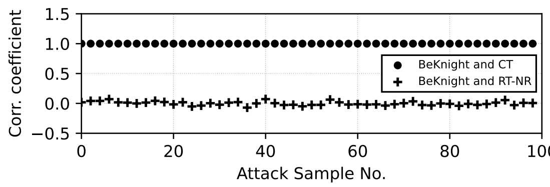
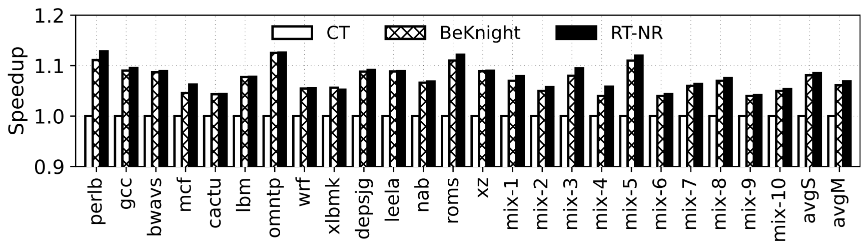
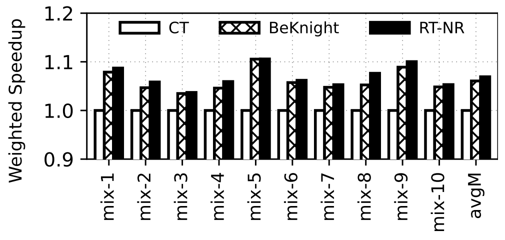


Fig. 10: Correlation of the attacker trace from BeKnight with traces from *secure* CT and traces from *unsecure* RT-NR.



(a) Non-SMT Configuration



(b) SMT Configuration

Fig. 11: Comparison of system performance (normalized to **CT**) for different schemes using ***history-based predictor***.

VIII. Evaluation

A. Security Analysis of BeKnight

We first discuss the security guarantees offered by BeKnight with respect to the attack anatomy (shown in Table III). As a different domain cannot observe the speculative state changes (which are stored in SPLB and not from another domain), *direction*-based PHT leakage is prevented. Meanwhile, since attacker only observes architectural states of PHT entries, it cannot infer whether a PHT entry has been speculatively updated or not, hence annulling *execution* based PHT leakage. Besides mitigating non-SMT attacks, BeKnight tracks speculative PHT updates from branch resolution, regardless of whether they will be committed or squashed. This ensures that IS-Unsafe entries are tracked in SPLB, preventing SMT attacks. Finally, *GHR leakage* inherently relies on observation through PHT, by ensuring no impact of speculation is visible to attacker through PHT, the GHR leakage is also defeated.

To verify the effectiveness of BeKnight in preventing BPU speculation leakage, we conduct a security analysis using a proof-of-concept cross-domain attack [13]. In this attack scenario, a victim executes a branch (b_v) whose outcome depends on a speculative secret accessed in the wrong execution path (Listing 1). The attacker attempts to infer the speculative **secret** by observing the prediction made for its own congruent branch (b_a). The results of the attack analysis are shown in Figure 9. Under the insecure *RT-NR* scheme (used in modern processors), the attacker successfully recovers the secret by observing the execution latency of b_a, which perfectly matches the corresponding secret values (Figure 9a). However, when BeKnight is enabled, the attacker's attempts to probe the PHT using b_v to infer the speculative update are thwarted by the domain conflict detection mechanism (Figure 9b). As a result, BeKnight obfuscates the prediction behavior of b_a, rendering it independent of the victim's speculative secret. The attacker's observations from the branch predictor become random, making it impossible to recover any secret information. Furthermore, correlation analysis on attack traces obtained from BeKnight against traces from CT and RT-NR confirm BeKnight offers same PHT states for cross-domain accesses as secure CT scheme (Figure 10).

B. Performance Evaluation

Mispredictions Per Kilo Instructions. MPKI (i.e., mispredictions per kilo instructions) is the critical metric for evaluating branch predictor design. Our results show that the **CT** scheme has the highest MPKI at 17.5 on average over all workloads. In contrast, the other two schemes utilizing speculative update of the PHT at resolution time achieve lower MPKI of 12.6 on BeKnight and 11.9 on RT-R. Additionally, we observe that single-program workloads generally have lower MPKI (on average, 11.4 to 8.7 across all schemes) whereas multi-program workloads show considerably higher MPKI (on average, 23.6 to 15 across all schemes). We note that this is because of PHT conflicts–PHT entry trained in one domain does not help the prediction of a congruent branch in another domain.

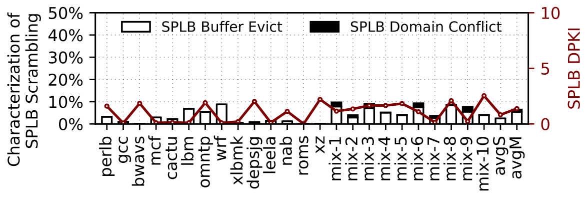


Fig. 12: Breakdown of BeKnight SPLB discard per squash of resolved branches and DPKI (discard per kilo instructions).

Application Performance of BeKnight Schemes. Figure 11a shows the overall system performance (i.e., instructions per cycle) in various schemes. Similar to the MPKI, it is evident that speculative update of the PHT based on resolved branch has superior performance over commit-time update. Specifically, RT-NR has 8.5% and 6.8% performance gain over CT for single- and multi-programmed workloads respectively. In contrast, with the SPLB-based speculative leakage prevention mechanism, BeKnight can achieve 8.1% performance gain over CT for single threaded workloads and 6.1% for multi-programmed workloads. Moreover, BeKnight schemes perform very close to best-performing **RT-NR** with a minimal 0.3% and 0.8% performance drop in single- and multi-program workloads respectively. Figure 11b shows the performance of the schemes in SMT setup. Note that with the multi-program workloads in SMT, it is expected to have higher SPLB conflicts. Overall, BeKnight has 6% IPC improvement over commit-time and only 0.8% overhead on top of RT-NR.

Impact of Unsafe Tracking. We quantitatively analyze the performance of SPLB and its impact on system performance. Particularly, there are two cases when SPLB can directly influence the performance: 1) SPLB eviction for an unsafe entry because of limited capacity, and 2) SPLB domain conflict when an unsafe entry tracked in SPLB is accessed from a different domain (i.e., another thread). In both cases, BeKnight must

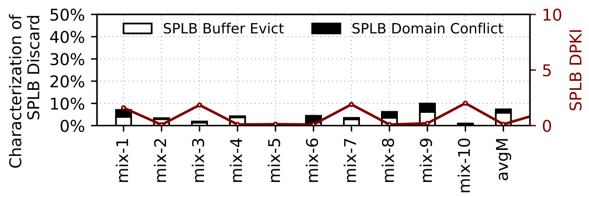


Fig. 13: Characterization of SPLB in SMT workloads.

Size	**Latency** (ns)	**Energy** (nJ)	**Area** (mm^2)	**Storage** (B)
64	0.067	0.0002	0.0003	184
128	0.071	0.0002	0.0004	352
256	0.085	0.0003	0.0006	672
512	0.087	0.0004	0.0013	1344

TABLE IV: Hardware and operational (latency and dynamic energy) overhead of BeKnight framework.

discard the *S* corresponding to the SPLB entry to eliminate the speculative state changes. Figure 12 shows the percentage of SPLB discards (normalized to number of resolved but squashed branches). We can see that single-program workloads have lower SPLB discard rate (2.6%) compared to that of RR-based multi-program workloads (6.1%). Additionally, there is no domain conflict in the single-program workloads (as all accesses are coming from the same domain). In contrast, RR-based multi-program workloads show 1% domain conflict and the rest (5.3%) due to eviction. The SPLB discard Per Kilo Instruction (SPLB DPKI) shows a moderate number of 0.75 and 1.36 SPLB DPKI. Compared to RR-based workloads, SMT-based workloads (Figure 13) have a slightly higher PHT discard rate due to increased domain conflicts (1.5%).

C. Hardware Overhead Analysis

We use CACTI 7.5 [38] to model SPLB to investigate its hardware overhead. Table IV lists the access latency, dynamic energy per access, area, and storage overhead for different SPLB sizes. Note that SPLB is always set to be *4-way* set associative. With the default SPLB size of 128, PHT offset requires 9 bits (for a 16K PHT) and the *Speculative counter* requires 4-bit. Along with 12-bit `tid`, its storage overhead is 432 bytes. The access latency for SPLB is well below the single cycle latency of modern processors. Therefore, its lookup/update procedures do not add any additional delay on the branch prediction path. Synthesized using 22nm process technology, the area overhead is insignificant compared to overall CPU die area (e.g., $160mm^2$ die area of Ivy Bridge i7 processor [39]). We additionally synthesize the branch predictor in *CT* and *BeKnight* using Synopsis Design Compiler with FreePDK45 standard library [40] and observe that BeKnight requires minimal 6.4mW additional dynamic power for one prediction, update, commit cycle of a branch.

IX. Related Works

Microarchitectural attacks, specifically transient execution attacks, present significant security risks to computing systems. These attacks exploit performance optimizations in modern processors to leak speculatively accessed secrets through side channels. Existing transient execution attacks often rely on BPU to trigger speculation [1], [9], [11] and utilize BPU [9], [13], [27], caches [1], [12] and other components [11] for secret transmission. Hardware-specific defensive techniques have been proposed to counteract transient execution attacks, with a focus on hiding or cleaning the speculation footprint on caches [41]–[43]. Oblivious speculation techniques [18]–[21], [43] have also shown promise in mitigating speculation side channels by delaying the impact of speculative execution on microarchitectural components. However, investigating hardware-specific defenses is crucial to leverage the performance characteristics of individual microarchitecture components. This work introduces novel schemes that harness the positive performance impact of early BPU pattern history updates while maintaining security.

Architecture-level techniques for defending against microarchitectural side channels typically employ resource isolation and obfuscation to prevent the observation of microarchitectural states [44]–[48]. Specifically, existing works have explored secure branch predictors to mitigate BPU as a general source of side channels [14]–[17], [49], [50]. Some approaches enforce isolation of branch predictor buffer structures among security contexts [14], leverage obfuscation through randomization and content-encoding of BPU indexing [16], or implement strict hardware isolation for critical BPU structures [17], [49]. While these methods can mitigate BPU leakage in both speculative and non-speculative domains, they face scalability issues and performance impacts due to the need for separate BPU structure copies per context and runtime state purging. BeKnight, effectively prevents BPU leakage in the speculative domain with minimal overhead by providing efficient security domain tracking and leakage-free restoration.

X. Conclusion

In this paper, we present BeKnight, a novel protection framework that secures speculatively-updated branch predictors against information leakage in the speculation domain. BeKnight is designed with the aim to retrain performance benefits of early BPU state (i.e., pattern history) updates while ensuring its security in the speculation domain. BeKnight employs a very small pattern lookaside buffer to track the ownership of unsafe PHT entries among many security domains. By keeping the speculative updates separate in SPLB, BeKnight maintains speculative predictor update behavior for same-domain while ensuring predictions are made using architecturally correct states for cross-domain access. Our evaluation results show that BeKnight can completely defeat the speculative side channel on BPU while performing almost the same as the insecure high-performance baseline.

Acknowledgements

This work is supported in part by the U.S. National Science Foundation under award numbers CNS-2008339 and CNS-2147217. Part of the results presented in this paper were obtained using the Chameleon testbed supported by NSF.

References

[1] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *IEEE S&P*, 2019.

[2] D. Evtyushkin, R. Riley, N. Abu-Ghazaleh, and D. Ponomarev, "BranchScope: A new side-channel attack on directional branch predictor," in *ACM ASPLOS*, 2018, pp. 693—-707.

[3] O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert, "Predicting secret keys via branch prediction," in *Cryptographers' Track at the RSA Conference*. Springer, 2007, pp. 225–242.

[4] O. Acıiçmez, S. Gueron, and J.-P. Seifert, "New branch prediction vulnerabilities in openssl and necessary software countermeasures," in *IMA International Conference on Cryptography and Coding*. Springer, 2007, pp. 185–203.

[5] F. Yao, G. Venkataramani, and M. Doroslovački, "Covert timing channels exploiting non-uniform memory access based architectures," in *ACM GLSVLSI*, 2017, pp. 155–160.

[6] D. J. Bernstein, "Cache-timing attacks on aes," 2005.

[7] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Understanding and mitigating covert channels through branch predictors," *ACM TACO*, vol. 13, no. 1, pp. 1–23, 2016.

[8] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack directories, not caches: Side channel attacks in a non-inclusive world," in *IEEE S&P*, 2019, pp. 888–904.

[9] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside sgx enclaves with branch shadowing," in *USENIX Security*, 2017, pp. 557–574.

[10] A. Mambretti, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, and A. Kurmus, "Two methods for exploiting speculative control flow hijacks," in *USENIX WOOT*, 2019.

[11] W. Xiong and J. Szefer, "Survey of transient execution attacks," *ACM Computing Surveys*, 2021.

[12] T. Zhang, K. Koltermann, and D. Evtyushkin, "Exploring branch predictors for constructing transient execution trojans," in *ACM ASPLOS*, 2020, pp. 667–682.

[13] M. H. I. Chowdhuryy and F. Yao, "Leaking secrets through modern branch predictor in the speculative world," *IEEE Transactions on Computers*, 2021.

[14] I. Vougioukas, N. Nikoleris, A. Sandberg, S. Diestelhorst, B. M. Al-Hashimi, and G. V. Merrett, "Brb: Mitigating branch predictor side-channels." in *IEEE HPCA*. IEEE, 2019, pp. 466–477.

[15] T. Zhang, T. Lesch, K. Koltermann, and D. Evtyushkin, "Stbpu: A reasonably safe branch predictor unit," *arXiv preprint arXiv:2108.02156*, 2021.

[16] J. Lee, Y. Ishii, and D. Sunwoo, "Securing branch predictors with two-level encryption," *ACM TACO*, vol. 17, no. 3, pp. 1–25, 2020.

[17] L. Zhao, P. Li, R. Hou, M. C. Huang, X. Qian, L. Zhang, and D. Meng, "Hybp hybrid isolation randomization secure branch predictor," in *IEEE HPCA*, 2022.

[18] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, "Speculative Taint Tracking (STT) A Comprehensive Protection for Speculatively Accessed Data," in *IEEE MICRO*, 2019, pp. 954–968.

[19] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci, "NDA: Preventing speculative execution attacks at their source," in *IEEE ISCA*, 2019, pp. 572–586.

[20] P. Li, L. Zhao, R. Hou, L. Zhang, and D. Meng, "Conditional speculation: An effective approach to safeguard out-of-order execution against spectre attacks," in *IEEE HPCA*, 2019, pp. 264–276.

[21] K. Barber, A. Bacha, L. Zhou, Y. Zhang, and R. Teodorescu, "Specshield: Shielding speculative data from microarchitectural covert channels," in *IEEE PACT*. IEEE, 2019, pp. 151–164.

[22] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander, "Efficient invisible speculative execution through selective delay and value prediction," in *IEEE ISCA*, 2019, pp. 723–735.

[23] S. McFarling, "Combining branch predictors," Citeseer, Tech. Rep., 1993.

[24] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with {TLB} attacks," in *USENIX Security)*, 2018, pp. 955–972.

[25] F. Yao, M. Doroslovački, and G. Venkataramani, "Covert timing channels exploiting cache coherence hardware: Characterization and defense," *International Journal of Parallel Programming*, vol. 47, no. 4, pp. 595–620, 2019.

[26] F. Yao, M. Doroslovacki, and G. Venkataramani, "Are coherence protocol states vulnerable to information leakage?" in *IEEE HPCA*, 2018, pp. 168–179.

[27] M. H. I. Chowdhuryy, H. Liu, and F. Yao, "BranchSpec: Information Leakage Attacks Exploiting Speculative Branch Instruction Executions," in *IEEE ICCD*, 2020.

[28] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE symposium on security and privacy*. IEEE, 2015, pp. 605–622.

[29] W. Koch, "Mitigate a flush+reload cache attack on RSA secret exponents." [Online]. Available: https://github.com/gpg/libgcrypt/commit/e2202ff2b704623efc6277fb5256e4e15bac5676

[30] G. H. Loh, "Revisiting the performance impact of branch predictor latencies," in *IEEE ISPASS*, 2006, pp. 59–69.

[31] K. Skadron, M. Martonosi, and D. Clark, "Speculative updates of local and global branch history: A quantitative analysis," *Journal of Instruction-Level Parallelism*, vol. 2, 2000.

[32] E. Hao, P.-Y. Chang, and Y. N. Patt, "The effect of speculatively updating branch history on branch prediction accuracy, revisited," in *IEEE MICRO*, 1994, pp. 228–232.

[33] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. Fletcher, "Speculative taint tracking (stt): A formal analysis," *University of Illinois at Urbana-Champaign and Tel Aviv University, Tech. Rep*, 2019.

[34] N. Soundararajan, S. Gupta, R. Natarajan, J. Stark, R. Pal, F. Sala, L. Rappoport, A. Yoaz, and S. Subramoney, "Towards the adoption of local branch predictors in modern out-of-order superscalar processors," in *IEEE MICRO*, 2019, pp. 519–530.

[35] M. Yan, J.-Y. Wen, C. W. Fletcher, and J. Torrellas, "Secdir: a secure directory to defeat directory side-channel attacks," in *IEEE ISCA*, 2019, pp. 332–345.

[36] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM CAN*, vol. 39, no. 2, pp. 1–7, 2011.

[37] A. Fog, "The microarchitecture of Intel, AMD, and VIA CPUs: An optimization guide for assembly programmers and compiler makers," *Technical University of Denmark*, 2021. [Online]. Available: https://www.agner.org/optimize/microarchitecture.pdf

[38] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New tools for interconnect exploration in innovative off-chip memories," *ACM TACO*, vol. 14, no. 2, 2017.

[39] A. L. Shimpi, "Dual Core/GT2 Ivy Bridge Die Measured: 121mm2̂." [Online]. Available: https://www.anandtech.com/show/5875/dual-coregt2-ivy-bridge-die-measured-121mm2

[40] J. E. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W. R. Davis, P. D. Franzon, M. Bucher, S. Basavarajaiah, J. Oh, and R. Jenkal, "Freepdk: An open-source variation-aware design kit," in *IEEE MSE*, 2007, pp. 173–174.

[41] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "InvisiSpec: Making speculative execution invisible in the cache hierarchy," in *IEEE MICRO*, 2018, pp. 428–441.

[42] G. Saileshwar and M. K. Qureshi, "Cleanupspec: An 'undo' approach to safe speculation," in *IEEE MICRO*, 2019, pp. 73–86.

[43] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Safespec: Banishing the spectre of a meltdown with leakage-free speculation," in *IEEE DAC*, 2019, pp. 1–6.

[44] F. Liu and R. B. Lee, "Random fill cache architecture," in *IEEE MICRO*, 2014, pp. 203–215.

[45] F. Yao, H. Fang, M. Doroslovački, and G. Venkataramani, "COTSknight: Practical defense against cache timing channel attacks using cache monitoring and partitioning technologies," in *IEEE HOST*, 2019.

[46] H. Fang, S. S. Dayapule, F. Yao, M. Doroslovački, and G. Venkataramani, "Prefetch-guard: Leveraging hardware prefetches to defend against cache timing channels," in *IEEE HOST*, 2018, pp. 187–190.

[47] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *IEEE HPCA*, 2016, pp. 406–418.

[48] F. Yao, H. Fang, M. Doroslovački, and G. Venkataramani, "Leveraging cache management hardware for practical defense against cache timing channel attacks," *IEEE Micro*, vol. 39, no. 4, pp. 8–16, 2019.

[49] L. Zhao, P. Li, R. Hou, M. C. Huang, J. Li, L. Zhang, X. Qian, and D. Meng, "A lightweight isolation mechanism for secure branch predictors," in *IEEE DAC*, 2021, pp. 1267–1272.

[50] A. Mondelli, P. Gazzillo, and Y. Solihin, "Sempe: Secure multi path execution architecture for removing conditional branch side channels," in *IEEE DAC*, 2021, pp. 973–978.