HyBP: Hybrid Isolation-Randomization Secure Branch Predictor

Lutan Zhao[†], Peinan Li[†], Rui Hou[†*], Michael C. Huang[‡], Xuehai Qian[¶], Lixin Zhang[§] and Dan Meng[†]
[†]State Key Laboratory of Information Security, Institute of Information Engineering, CAS
and School of Cyber Security, University of Chinese Academy of Sciences.
[‡]University of Rochester. [§]Freelance. [¶]University of Southern California.
[†]Email: {zhaolutan, lipeinan, hourui, mengdan}@iie.ac.cn, *Corresponding author: Rui Hou
[‡]Email: michael.huang@rochester.edu, [¶] Email: xuehai.qian@usc.edu.

Abstract—**Recently exposed vulnerabilities reveal the necessity to improve the security of branch predictors. Branch predictors record history about the execution of different processes, and such information from different processes are stored in the same structure and thus accessible to each other. This leaves the attackers with the opportunities for malicious training and malicious perception. Physical or logical isolation mechanisms such as using dedicated tables and flushing during context-switch can provide security but incur non-trivial costs in space and/or execution time. Randomization mechanisms incurs the performance cost in a different way: those with higher securities add latency to the critical path of the pipeline, while the simpler alternatives leave vulnerabilities to more sophisticated attacks.**

This paper proposes HyBP, a practical hybrid protection and effective mechanism for building secure branch predictors. The design applies the physical isolation and randomization in the right component to achieve the best of both worlds. We propose to protect the smaller tables with physically isolation based on (thread, privilege) combination; and protect the large tables with randomization. Surprisingly, the physical isolation also significantly enhances the security of the last-level tables by naturally filtering out accesses, reducing the information flow to these bigger tables. As a result, key changes can happen less frequently and be performed conveniently at context switches. Moreover, we propose a latency hiding design for a strong cipher by precomputing the "code book" with a validated, cryptographically strong cipher. Overall, our design incurs a performance penalty of 0.5% compared to 5.1% of physical isolation under the default context switching interval in Linux.

I. Introduction

Recently exposed vulnerabilities reveal the necessity to improve the security of branch predictors in mainstream commercial processors [1]–[6]. The root cause of these vulnerabilities is that modern processors generally adopt the design principle of resource sharing, and branch predictor is a typical example. From a security perspective, resource sharing leads to a possible attack surface. Branch predictors record history about the execution of different programs, and such information is stored in the shared structure and thus accessible to all processes. This leaves the attackers the opportunities for malicious training and/or perception.

The fundamental defense strategy is to provide an isolated execution environment for sensitive processes. Software mitigations (*e.g.*, converting secret-dependent branch instructions into indirect jumps/computation instructions automatically [7], and inserting specific instructions manually to protect high privileged processes from malicious training by lower privileged processes [8]) cannot eliminate all runtime sensitive information leakage. Hardware-assisted isolation can be classified into two categories: 1) Flushing the whole history table in a context switch or privilege change. Software implementations typically introduce significant slowdowns [9], which can be reduced by the hard-assisted flush [10], [11]. Yet, because flushing only happens during context or privilege switches, it can not completely solve the problem in SMT architectures. 2) Physical isolation of context from different threads and privilege levels. BRB [12] is a state-of-the-art hardware implementation that provides individual history tables for different programs. Although BRB tries to limit hardware cost, it needs maintain separate tables for all software thread-privilege level combinations. Such physical isolation is fundamentally insufficient: with limited storage, it can only provide isolation for limited number contexts or threads.

Another promising solution is to apply randomization to branch predictors [13]–[15]. The Samsung Exynos CPU has implemented content encryption via simple substitution cipher in branch-target buffers (BTB) and return address stack (RAS) [13], but it only protects against some Spectre variants (*e.g.*, Spectre V2 and Spectre RSB), and lacks effective coverage to other side channel attacks. Zhao et al. employs lightweight XOR operation to encode the content and index to mitigate branch predictor side-channels [15]. Lee et al. also propose to randomize the index of branch predictor using low-latency cipher [14]. However, both simple XOR operation and LLBC proposed by CEASER [16] have been proved to be linear and vulnerable to cryptanalytic attacks—the complexity of finding an eviction set is the same as when there is no randomization present [17], [18].

For a randomization based protection, encryption is the crucial component that determines how secure the scheme is. For branch predictor, the cipher needs to be embedded into the pipeline, leading to important trade-offs between security and performance. In modern branch predictor, the prediction latency is usually 2 to 3 cycles. However, a typical strong cipher takes many more cycles to produce the ciphertext (*e.g.*, over 10 cycles in AES, 8 cycles in PRINCE on a 4GHz processor [19]). Thus, a key challenge is how to embed it into the pipeline without major performance degradation. The current solutions essentially choose to use the simple cipher to trade-off security for low performance overhead. Making it worse, a randomization-based secure branch predictor using a strong cipher (e.g. AES cipher) still requires changing the key more than one hundred times in a typical Linux OS time slice. Lowering the change frequency will significantly increase the probability of a successful attack within a time slice.

In summary, the existing defense mechanisms suffers from one of the three drawbacks: *inadequate*—flush cannot protect different threads in SMT processor; or *ineffective*—physical partition or replication incurs very high overhead and considerable performance degradation; or *expensive*—randomization is not panacea, and incurs high overhead to be secure. We implemented existing schemes on the cycle-accurate Gem5 [20] simulator and show the cost and performance trade-offs in Table I. The *flush* is performed on context switch and privilege change, and can be finished in one cycle, but it is not useful for SMT when multiple threads execute simultaneously. In *partition*, we fix the size of branch predictor and divide it into four partitions (for SMT-2, two threads, each has user and kernel mode). Each partition is flushed on context switch. The performance loss is due to smaller partition for each thread for a given mode. In *replication*, we first increase the size of branch predictor proportionally with the number of threads, and then partition it according to (thread, privilege) combinations. For SMT-2, the overhead is 100% and the performance loss is due to smaller portion for each (thread, privilege). *Disable SMT* shows the performance loss of not being able to execute two threads simultaneously. The results will be discussed in detailed in Section VII.

Defense Mechanism	**Performance overhead**	**Hardware cost**	**Security**	
			Single-Thread	**SMT**
Flush	5.1%	0	✓	✗
Partition	6.3%	0	✓	✓
Replication	2.1%	100%	✓	✓
Disable SMT	18%	0	–	✓
HyBP	**0.5%**	**21.1%**	✓	✓

Table I: Comparison of security mechanism.

While the performance loss of the alternative mechanisms seems to be not quite significant, but we *should not take the single digit percentage lightly for the general-purpose processors*. They run all workloads, not only the benchmarks but also the unforeseen ones. Thus, this paper's goal is to develop a new secure branch predictor *with minimum performance loss with reasonable hardware overhead*. Given the analysis of existing schemes, achieving this goal requires fundamental innovations.

We propose HyBP, a practical hybrid protection and effective mechanism for building secure branch predictors. The *key insight* is that, we can *apply the physical isolation and randomization in the right component to achieve the best of both worlds*. Typically, modern branch predictors have multiple tables and hierarchical designs with upper-level small tables for best timing at low latency, and last-level large tables to increase prediction accuracy [13], [21]–[23]. We propose to protect the smaller tables with physically isolation based on (thread, privilege) combination; and protect the large tables with randomization. Intuitively, the design choice is very reasonable: physically isolating small tables incurs much less overhead compared to applying isolation for all tables. What's less intuitive is that, the physical isolation also significantly enhances the security of the last-level tables by *naturally filtering out accesses, reducing the information flow to these bigger tables*. As a result, key changes can happen less frequently and be performed conveniently at context switches. Moreover, we propose a latency hiding design for a strong cipher. By precomputing the "code book" with a validated, cryptographically strong cipher, it can be removed from the timing critical path—significantly mitigating the latency impact of using a strong cipher (*e.g.*, 8 in QARMA [24], PRINCE [19] on a 4GHz processor). Without sacrificing performance, this optimization provides stronger security than simple XOR operation and low-latency block-ciphers for which the cryptographic weakness can lead to a complete security subversion [17].

II. Background

A. Attacks Through Branch Predictors

Conventional branch predictors allow different processes to use the shared hardware resources for branch prediction. This creates a side channel similar to those exploited in a cache based side channel attack. It allows an attacker to prime the predictor in a certain way to leak victim's sensitive information. The attacker can also achieve malicious training in order to influence the victim's (speculative) execution, which in turn enables or enhances the victim's information leakage. Two types of attacks have been developed.

Reuse-based attacks: In structures such as PHT (Pattern History Table), different programs directly access the common resource. Entries set by one process may influence the other. This type of attack (*e.g.*, BranchScope [2], Spectre V1,V2 [1], and Branch Shadowing [10]) is therefore analogous to reuse-based cache attacks [25].

Contention-based attacks: In cache-like structures (*e.g.*, BTB), an attack similar to contention based cache attack can be mounted. One requirement for constructing such attacks is to create contention so that the old record is evicted. Then, an attacker learns about the execution of the target branches of a victim by sensing whether contention exists or not for the corresponding entry of a branch predictor table [5], [26].

B. *Randomization-Based Protection*

Compared to the intuitive mechanisms physical isolation and flushing, randomization achieves isolation of the content via encryption of information with less performance costs [13]–[15]. Randomization transforms both index and content of predictor tables—BTB and PHT—with thread-specific encryption/encoding keys to obstruct malicious threads from accessing contents of other threads. The keys change under certain conditions so that countermeasures have insufficient time to succeed. Figure 1 shows a specific example with a “content key” and an “index key” that we use in our design.

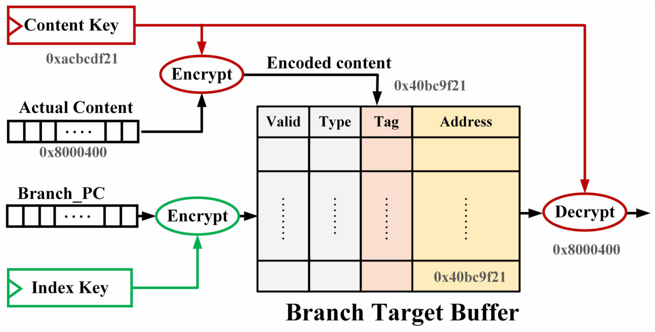


Figure 1: Overview of content encryption and index encryption. The red modules are designed for content encryption; The green ones are for index encryption.

The purpose of the encryption is to introduce randomization to the actual index and content used in BTB and make it harder for other entities to obtain useful information despite the storage sharing. When a different thread i (with its own set of keys) executes a branch, it will not obtain the original tag or target address updated by thread j due to the difference in their keys. It provides a logical isolation of the content of BTB among different threads and different privileges. The same randomization mechanism can also be applied to PHT.

III. Challenges of Randomization

Randomization breaks the fixed one-to-one correspondence between the original value of index or stored content and the actual value used in the predictor tables. This makes it harder, but not impossible, for the attacker to perform attacks. More importantly, it incurs high performance overhead, as we discuss below.

A. *Insecure Simple Ciphers*

A secure encryption algorithm should satisfy two requirements: 1) uniformity of the output space, which equalizes the probability that the output index falls on each branch predictor table entry; and 2) nonlinearity between the inputs and outputs, which is crucial to resist cryptanalytic attacks. The two requirements break the fixed index mapping relationship and significantly improve the difficulty of the attacker constructing conflicting items of a target branch instruction or performing malicious training of a target branch instruction. While simple low-latency ciphers such as CEASER’s LLBC [16] can produce a ciphertext in just 2 cycles, Purnal et al. [17], [18] revealed that cryptographic weaknesses in the CEASER’s LLBC can lead to a complete security subversion.

B. *Performance Overhead of Encryption*

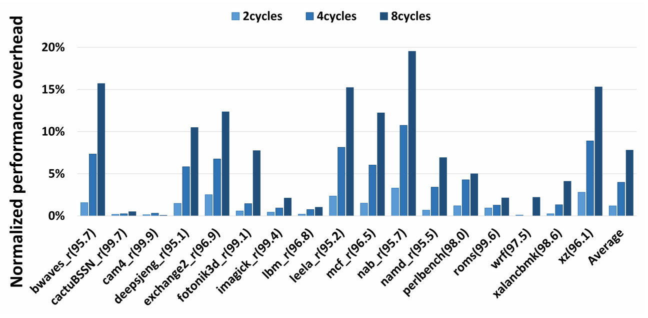


Figure 2: Performance impact on increased front-end pipeline due to encryption latency. (The number in parentheses is the prediction accuracy of each application.)

Encryption increases the number of cycles of the front-end pipeline, leading to the increased the penalty of mispredictions. We investigate such performance impacts on our evaluation platform, which is described in Section VII. Figure 2 shows the performance loss when the front-end pipeline is increased by 2, 4, and 8 cycles on a single-threaded core. As expected, the applications whose performance is the most sensitive to the longer pipeline are those with relatively low prediction accuracy. We see that with an encryption latency of 8 cycles, the performance loss can be as high as 19.5%. Overall, using a commodity cipher (*e.g.*, QARMA and AES) with a latency of 8 cycles, the average performance loss due to the longer front-end pipeline is close to 7.8%. The results suggest that simply using strong cipher is not feasible.

C. *Performance Overhead of Key Change*

Ideally, given any input, the randomization process creates a uniform distribution in the entire range in its output. It minimizes information on the input an attacker can obtain from observing the output. However, with repeated observations, an attacker may still succeed in extracting information. For example, an attacker can efficiently mount a contention-based attack against a particular branch, provided a small set of lines that map to the same set can be found so that executing these can create an eviction event for the target branch. The Group-Elimination Method (GEM) is a typical algorithm to create such an eviction set with high efficiency [27], even though GEM is not the state-of-the-art algorithm. The key point is that, it does not take a long time to break the randomization mechanism on BTB.

Given L random lines that conflict with a target, GEM excludes a group of lines at a time and retests to see if there is a conflict. If the conflict persists, the group of lines are known not to cause the conflict and can be eliminated. It can be shown that in $O(L)$ time, the algorithm will produce an eviction set [27]. If we assume that the GEM method works on BTB with the same efficiency, with a BTB size of 7K entries, about 2^{16} accesses are needed to construct an eviction set. This means that a key change is required at about every 2^{16} accesses, equivalent to flushing the table more than a hundred times in a typical operating system time slice. The performance impact will be significant. With smaller BTBs, the change needs to be even more frequent.

IV. Threat Model

This paper has the following assumptions: The attacker thread and the victim thread can run on the same processor core. An attacker can know the source code and address layout of the victim. An attacker has the ability to run the victim program in single-step mode, such as manipulating the APIC timer exploited in SGX-Step [28].

Table II: Classification of threat models. Our target scenarios are marked with ✓. Scenarios not considered are marked with ○. “privilege” is abbreviated as “priv”.

	Combinations of context switch and privilege change			
	Same-thread Same-priv	**Same-thread Cross-priv**	**Cross-thread Same-priv**	**Cross-thread Cross-priv**
Reuse-based (e.g. Bluethunder[20] Branchscope [2])	○	✓	✓	✓
Contention-based (e.g. Jump over ASLR [29])	○	✓	✓	✓

This paper focuses on defending against reuse-based and contention-based attacks on branch predictors. HyBP defend against most scenarios such as typical cross-thread or cross-privilege reuse-based attacks (e.g. Spectre V2 [1], Branch-Scope [2] and Bluethunder [3]) and contention-based attacks (e.g. Jump over ASLR [29]). The only exception is the same-thread and same-privilege attacks (e.g. Spectre V1 [1] and malicious trojan injection attacks [6]). However, we believe its root cause is not whether branch predictors is secure and they can be defended by other security mechanisms (e.g. SABC [30]). Therefore, we do not consider defending against such attacks, and we don’t think that secure branch predictors needs to solve this problem.

V. Hybrid Protection Mechanism

A. Defense Strategy

Based on the previous analysis of existing mechanisms, we believe a practical defense mechanism for branch predictor to isolate threads and privilege levels should satisfy three desirable properties:

- *Limited performance impact;*
- *Minimal impact on processor pipeline;*
- *Compatible with modern branch predictor structures.*

To achieve these goals, we propose HyBP, a practical and effective hybrid protection mechanism for building secure branch predictors. We now discuss the implementation issues. It features three key ideas:

Hybrid protection: We overcome the drawbacks of physical isolation and randomization by applying each mechanism in the right component to achieve the best of both worlds. Modern predictors consist of multiple tables and hierarchical designs with small upper-level tables and a large last-level table [21], [22]. We propose a *hybrid protection* mechanism that protect the smaller tables with physically isolation based on (hardware thread, privilege) combination; and protect the large tables with randomization.

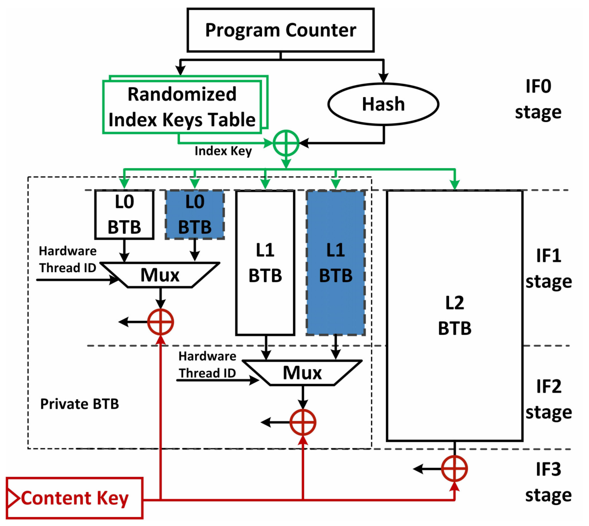
Strong encryption: We transform both index and table content with software thread-specific encryption/encoding keys to prevent malicious threads from accessing contents of other threads; the keys change under certain conditions so that countermeasures have insufficient time to succeed. While even simple encryption can defeat some existing attacks, more sophisticated attacks are found to defeat very recent proposals. We thus use strong encryption algorithms and update the key periodically.

Latency hiding: To reduce the performance loss of using strong encryption, of which the latency required is significant (*e.g.*, 8 cycles for QARMA [24], PRINCE [19]), we propose to store precomputed partial results to allow accelerated encoding/decoding in the instruction pipeline.

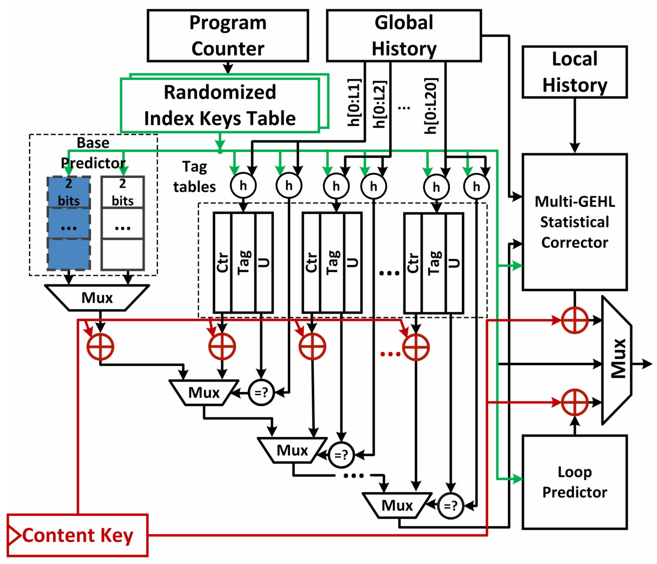
B. Isolation-Randomization Hybrid Scheme

The modern branch predictors [21], [22] have multiple tables and hierarchical design. Based on such architecture, neither physical isolation nor randomization is ideal: the small table sizes lead to a small number of accesses for an attacker to succeed; while physical isolation always incurs high hardware overhead. On the other side, the small table makes physical isolation relatively inexpensive to implement. The large tables can be protected by randomization (encoding), which can be made more secure when many accesses are naturally filtered out by the small tables.

Figure 3 shows an example of secure branch predictor design based on a baseline BPU from AMD Zen2 [21]. In the figure, the two upper-level BTBs (L0 and L1) and the base predictor in the TAGE direction predictor are protected with physical isolation. This can be achieved with a combination of replication and physical partition. For instance, the tables can be replicated for each SMT thread, with each thread only accessing its own table. In all cases, when a context switch occurs, the part of the table that belongs to the thread being swapped out is flushed. This can be easily achieved by changing the content and index keys. The larger tables (*e.g.*, L2 BTB) are protected by logical isolation with encoding. In all cases, the indices and content are encoded with index keys and content keys respectively.



(a) BTB with HyBP



(b) TAGE-SC-L predictor with HyBP

Figure 3: Example implementation of *hybrid* secure three-level BTB (a) and direction (TAGE-SC-L) predictor (b) on SMT-2 core. Each hardware thread has a randomized index keys table with as many as entries as the number of L2 BTB's sets or the entries of the longest tag table in TAGE predictor. Moreover, all the tables here are shown to use the shared content key, though using table-specific key is a valid option. The shaded tables (lower level BTBs in (a) and base predictor in (b)) are protected with physical isolation. The baseline predictor is from an AMD processor [21]: The three levels of BTB hold 16, 512, and 7K entries respectively, and the size of each entry is 60 bits, using random replacement; The TAGE component consists of a base predictor which is a simple PC-indexed 2-bit counter bimodal table (8Kbits prediction, 4Kbits hysteresis), thirty equal-sized tagged tables each with two interleaved banks featuring respectively (ten 12-bit 1K-entry banks, and twenty 16-bit 1K-entry banks); Each tagged entry consists a signed counter *Ctr* which sign provides the prediction, a (partial) *Tag* and an unsigned useful counter *U*. The respective tag widths are 8 and 11 bits. Other details can be found in [31].

The hybrid design achieves a good balance of security and implementation cost. Smaller tables gain good security protection with physical isolation, while the bigger tables avoid high cost in replication or physical partition. Interestingly, there is an additional security benefit for the bigger tables in a hybrid designs: L0 and L1 BTB naturally filter out accesses, making it more difficult for the attacker to observe a particular entry of the victim or to plant entries in the shared L2 BTB. Taking the PPP algorithm as an example [17], the algorithm relies on observing conflicts with a target line from the victim to perform its elimination process. In the presence of L0 and L1, the target line may not be present in the shared L2 table to begin with. Only through repeated evictions (due to activities from the victim) can the target line have a chance to enter the L2 table. If the system has an L0 and L1 BTB with physical isolation, when the victim executes a taken branch, the last-level BTB is no longer guaranteed to contain the line. Assuming the lower level tables have a miss rate of m, then the information flow to the last-level BTB is reduced to m times that without the L0 and L1. Roughly speaking, an attack algorithm will take $1/m$ times longer to obtain the same information, allowing the key to be changed less frequently.

C. Latency Hiding

So far, we have treated the encryption process as a black box. In a real design, the latency of encryption presents a challenge as discussed in Sec. III-B. While it is ideal that faster ciphers with sufficient security guarantees can be developed, we believe that it is very challenging and such cipher is not available at present. In HyBP, we choose to employ a fully validated commodity cipher, QARMA [32]. To mitigate the latency impact of using the cipher, we propose to hide much of the latency through precomputation of the "code book".

For index, we use a table to store a number of index keys generated by the commodity cipher, as shown in Figure 3. This table is indexed by a part of the branch's PC to retrieve an index key for that PC. The key is used together with the plaintext index to generate the actual index for table accessing.

For content encoding, we choose to use a simple XOR encryption for two reasons: 1) XOR operations are extremely fast with a trivial hardware cost; and 2) since the bit width of the content is much wider than the bit width of the index, changes of content keys makes a simple XOR encryption an effective mitigation technique (discussed in Section VI).

1) Precomputation for Keys Table: The production of the code book is left outside the critical path as shown in Figure 4 (shaded portion). When a new book is needed (*e.g.*, upon context switch), we use the encryption engine QARMA with an *Index Seed* to encode a sequence of readouts from the timer register. The resulting sequence of ciphertext is used to fill the code book. The *Index Seed* is calculated with the combination of ASID (Address Space ID), VMID (Virtual Machine ID) and RAND (value generated from random number generator or PUF [33], [34]). This index seed is generated completely in hardware, with no software visibility of any intermediate values, even to the hypervisor. We maintain an access counter for the code book. For security, the code book is renewed upon in two situations when 1) a context switch happens; and 2) the number of accesses based on the counter reaches a threshold.

The *Randomized Index Keys Table* is built with SRAM arrays. Its physical structure is irrelevant to the logical meaning of keys. For instance, a 1K-entry table with 10-bit keys per entry can be organized as a 256-entry array of 40-bit words. During a renew of the code book, after the initial pipeline fill-up of the cipher engine (of, say, 7 cycles), a single 40-bit key will be produced every cycle. Thus the code book will be renewed in 263 cycles.

2) Security Analysis of Keys Table: It is important that the keys table does not introduce new side channels or speculative execution vulnerabilities associated with the access of the table. We can see that for each BPU access, a keys table access is needed. The table is not a cache, there is no miss, and the access time is fixed. Therefore, the keys table itself does not have timing side channels. In addition, keys table renew is not affected by speculative execution or disrupted by interrupts or exceptions, so there is no speculative execution vulnerability. What's more, refreshing randomized index tables has two steps. The first is to update the content keys stored in registers (1 cycle), and then the keys table stored in the SRAM is refreshed (hundreds of cycles). Once the content key is updated, the encrypted entries touched by other previous threads cannot be decrypted correctly, thereby ensuring security.

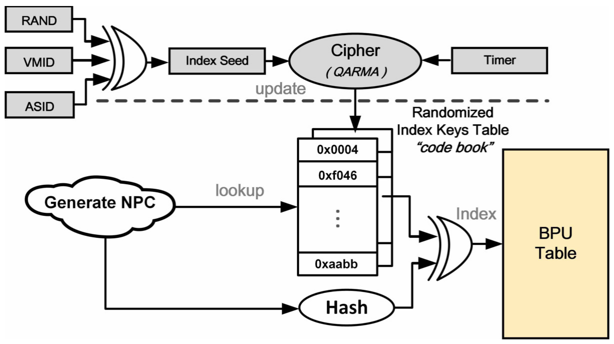


Figure 4: Overview of precomputation of the "code book" to mitigate the latency impact of using the QARMA. The gray shadow parts are for the generation of index keys.

D. Periodic Key Changes

The security of the system depends on a number of things, including the size of the table, the hierarchy, the attacking method, and the frequency of key changes. Given a particular design, the frequency of key changes becomes the primary control knob for security. Taking the baseline BTB as an example, with L0 and L1 using physical isolation, and a last-level table of 1024 sets, it requires roughly 2^{28} accesses to construct the PPP-based attack [17]. Thus, a key change on the order of 2^{28} instructions can be used.

Since the time period for change is comparable to that of context switch time, it is advantageous to initiate key changes at context switches. Intuitively, as a thread is switched in, there is minimal amount of its state left over in the tables, and thus a key change at this point will provide the least amount of performance impact.

Additionally, to prevent cross-thread or cross-privilege level attacks, each (thread, privilege) combination has its own set of keys in the *randomized index keys table*. In other words, the table holding keys for randomization is physically isolated among different active threads (in SMT) and privilege levels. When there is a context switch, the current thread's keys in the randomized index keys table are updated. It should be noted that the number of keys is non-trivial (*e.g.*, 2K entries in our example). Even with pipelining and wide datapath encrypters, it requires a noticeable amount of time to finish the update. Fortunately, the nature of our application of encryption suggests that even if an instruction uses an old key during the update procedure, the impact is only on its prediction accuracy instead of the final correctness. Therefore, we do not stall the execution pipeline during the update. We evaluated this performance loss in Section VII-E.

VI. Security analysis

Table III summarizes the security of different defense mechanisms. Flush alone cannot completely defend against contention-based attacks in SMT core. Physical isolation can also be improved due to higher storage overhead (multiplexed in time). In case of BRB, its storage overhead is more than twice that of HyBP. In contrast, hybrid isolation mechanisms are both more secure than these defense mechanisms. The specific analysis is detailed as below.

Table III: Summary of the Protections.

Defense Mechanism		**Single-threaded core**		**SMT core**	
		Reuse	**Contention**	**Reuse**	**Contention**
BTB	Flush	*Defend*	*Defend*	*No Protection*	*No Protection*
	Physical Isolation	*Defend*	*Defend*	*Defend*	*Defend*
	HyBP	*Defend*	*Defend*	*Defend*	*Defend*
PHT	Flush	*Defend*	*Defend*	*No Protection*	*Defend*
	Physical Isolation	*Defend*	*Defend*	*Defend*	*Defend*
	HyBP	*Defend*	*Defend*	*Defend*	*Defend*

A. Security of BTB with HyBP

In a hybrid protection mechanism, a upper-level BTB is physically isolated. Upon context switches, the previous history becomes unrecognizable in the latter thread. Thus the attacker cannot steal secret from a upper-level BTB. The security vulnerabilities remain in the shared BTB. In this analysis, we assume that the last-level BTB is *W*-way set associative with an *S*-bit set index and a *T*-bit tag per entry. Their typical values will be drawn from AMD Zen2 [21].

1) Reuse-based attacks: During such an attack, the entry left by one party is being translated in multiple ways before being used by another party, which makes it exceedingly difficult to maintain controlled manipulation. Taking malicious training as an example, the attacker wants to lay traps in the BTB to direct the victim to a meaningful location. First, these entries need to result in a BTB hit to be even considered. For that the (partial) tag left by the attacker has to match the encoded tag of a victim branch. The chance for one entry to have a BTB hit is $1/2^T$. The attacker can certainly lay many such traps, increasing the overall probability to some degree. But there is a second hurdle to clear: The content of the entry will need to lead to a meaningful location, one that contains the malicious code. Since the content is XOR-ed with another unknown key, the probability of leading the victim to a specific address is $1/2^N$, N being the number of stored partial address bits. Again, the attacker can certainly prepare many such traps and extend the attack over many intervals. But overall, the chance of success is against a very large denominator 2^{N+T}. And $N + T$ is usually more than 30 bits. Thus the time to perform reuse-based attack is greater than the OS time slice.

The general analysis applies to the case of an SMT core as well. The slight advantage to an attacker here is that they can continue to re-plant new traps in the BTB, whereas in a single-threaded core, those entries are gradually evicted and the attack strength reduces with time in a context switch interval.

2) Contention based attacks: In a contention-based attack carried out on a conventional core, an attack like Jump [29] can observe evictions of its own entry in BTB and thus infer the address of a taken branch executed by the victim. In our system, different hardware threads have different private keys for indexing. Without the key of the victim thread, the attacker can no longer make the inference of the branch address. However, the attacker can conceivably extend attacks on randomized caches to BTB. These attacks include the state-of-the-art technique for finding eviction set, Prime+Prune+Probe (PPP) [17] and blind contention.

PPP-Based Attacks.

Inspired by PPP algorithms, we developed an attack algorithm for the hierarchical BTB (shown in Algorithm 1). Note that CaSA also use the PPP algorithms as the first step in its end-to-end quantitative security analysis framework. Different from CaSA, we use a more rigorous security analysis because we assume that once a target eviction set is found, the attack is considered successful. We also assume that the attacker knows the virtual addresses of the program. Algorithm 1 uses binary search to minimize the number of victim invocations and adopting burst accesses and bootstrapping to optimize the total BPU accesses. We considered the worst case, assume that each branch instruction access to the branch predictors will change the state of the branch predictor, and branch predictor accesses occur every cycle, so as to evaluate the number of required accesses for a PPP-based attacks under extreme conditions.

The algorithm can be understood as going through three steps:

Step 1: Preparing candidate set. Since an attacker can control the virtual addresses, he can carefully construct candidate collection C and split it into S subsets C_i. Each subset contains W different elements. Note that each C_i is actually an potential eviction set. Elements in C_i are indexed to the same set in the original, un-randomized index, while C_i and C_j have different indexes. However, due to randomization, C_i and C_j may map to the same set in L2 BTB. These self-conflicts will be eliminated next.

Algorithm 1 Constructing Eviction Sets in HyBP

Input: *x*, victim target branch; *g*, victim gadget code; *S*, last-level BTB sets; *W*, last-level BTB ways; *C*, candidate set.
Output: Eviction set for *x*.

```
1: $\{C_1,C_2...C_S;|C_i| = W\} \leftarrow split(C,S)$
2: for $C_i$ in C do
3:     if $prune(C_i,C\backslash C_i)$ then
4:         $C \leftarrow C\backslash C_i$
5:     end if
6: end for
7: while $|C| > W$ do
8:     $\{G_1,G_2\} \leftarrow split(C,2)$
9:     if $\mathbb{E}(test(G_1,g\cup x)) > \mathbb{E}(test(G_1,g))$ then
10:        $C \leftarrow G_1$
11:    else if $\mathbb{E}(test(G_2,g\cup x)) > \mathbb{E}(test(G_2,g))$ then
12:        $C \leftarrow G_2$
13:    else
14:        return False
15:    end if
16: end while
17: return C
```

Step 2: Eliminating self-conflicts. In the subsequent *For* loop (Lines 2-6), the algorithm prunes the candidate set C by sequentially loading into the BTB each C_i and measuring the re-access delay. If a branch misprediction occurs during a subset C_i (delay is greater than a threshold), it means that one or more branches were evicted from BTB due to a set conflict and C_i needs to be removed from C.

Step 3: Binary search of conflict set. After pruning, collection C contains s' subsets ($s' \leq S$). The algorithm then uses binary search to find out which C_i conflicts with the target branch. In the *While* loop (Lines 7-16), the algorithm first splits the collection C roughly equally into G_1 (C_1, C_2... $C_{\lceil s'/2 \rceil}$) and G_2 ($C_{\lceil s'/2 \rceil+1}$, $C_{s'/2+2}$...$C_{s'}$). The *test* function determines whether the G_i conflicts with the victim. The algorithm measures delay of re-accessing G_i

after the victim performs a gadget code with and without x, respectively. If there is a perceived branch misprediction (longer delay) after the gadget with x but not after the gadget without x, this indicates that G_i is an eviction set of x. To increase confidence of the decision, the algorithm repeats the test and uses the mathematical expectation of the misprediction deviation between $test(G_i, g)$ and $test(G_i, g \bigcup x)$ to determine whether G_1 or G_2 contains the eviction set of x (lines 9 and 11). If there is not enough difference between the two cases, the search fails. Otherwise, the binary search repeats on the subset G_i that contains the conflict set.

Compared with the baseline, our hybrid protection mechanism increases the difficulty of constructing evictions sets. First, the physically isolated L0 and L1 BTB brings more uncertainty to testing the number of misses in lines 9 and 11 of the algorithm. Our experimental results show that the algorithm now has about 1% probability of success. Consequently, an attacker needs to run the algorithm many times. Concretely, for a 7-way L2 BTB with 2^{10} sets, constructing an eviction set for the target branch instruction (using the 1% success probability) requires profiling roughly 2^{27} BTB accesses. Second, the keys table for randomization will renew periodically. 2^{27} BTB accesses will take much longer than a typical context switch interval. Upon a context switch, the eviction set – even if found – is now obsolete.

Blind Contention Based Attacks. Another alternative intuitive method is to randomly select some sets to detect the target branch without finding the eviction set. To simplify the analysis, we assume that an attacker has the ability to completely filter out noise from other sets, but he still needs to eliminate the self-conflicting noise of the target set. In a BTB with S sets, there is a $1/S$ probability that a branch instruction falls in the same set of the target branch because the attacker and the victim use uncorrelated randomized index mapping. Suppose that the attacker employs n instructions to construct the blind contention and there are i instructions falling on the target set, the probability is$\binom{n}{i} \cdot \left(\frac{1}{S}\right)^i \cdot \left(1 - \frac{1}{S}\right)^{(n-i)}$. Since if $i > W$, there will inevitably be self-conflicts among them, thus i $\in [1, W]$ and the probability of the i instructions falling into W ways without self-conflict noise is $\frac{W!/(W-i)!}{W^i} \cdot \frac{i}{W}$. Finally, the probability that a valid conflict occurs on the victim's target branch instruction is given by:

$$P = \sum_{i=1}^{W} \left(\binom{n}{i} \cdot \left(\frac{1}{S}\right)^i \cdot \left(1 - \frac{1}{S}\right)^{(n-i)} \cdot \frac{W!/(W-i)!}{W^i} \cdot \frac{i}{W} \right) \tag{1}$$

For a BTB with 1024 sets and 7 ways, numerical analysis shows that the probability of a valid conflict is maximal (12%) when $n = 1140$. The number of expected accesses to probe one secret at a time is n/P. For the hybrid protection mechanism, the probability of the target branch locating at the last-level BTB is $1/(L0 \cdot L1)$. The probability that an attacker is able to sense the target branch is reduced to $1/(L0 \cdot L1)$ of equation (1). The attacker needs at least $n \cdot L0 \cdot L1/P$ accesses to successfully probe one target branch. At least 2^{28} accesses are required for one round of blind contention attack. For multi-bit keys, the success rate of blind contention based attacks will be further reduced. The probability of success of the attack stealing a 32-bit key is less than one in a million. Note that this is the probability in a completely noise-free situation, in fact the blind contention based attacks are very susceptible to lots of noise, therefore the attacker usually performs the eviction based attacks instead.

B. Security of PHT with HyBP

1) Reuse-based attacks: Suppose an attacker is powerful enough to filter out his own noise, and the attacker knows the table where the target branch locates. Security for TAGE-SC-L predictor with hybrid protection mechanism is analyzed as following.

First, the attacker cannot manipulate the victim deterministically to execute the wrong path speculatively due to content encoding (*e.g.*, tag and saturating counter). Second, index encoding poses strict requirements on attacks to perceive the direction of target branch. An attacker can no longer infer the direction through one Prime-Probe operation. The average number of accesses to perform an effective Prime-Probe operation is given by:

$$2^{I+T} \cdot \left(2^C + 2^U + 1\right) \tag{2}$$

I is the number of entries in each tag table, and U is the useful bit of tag entry. With $I = 13$, $T = 12$, $C = 2$, $U = 1$, the minimum number of accesses per cycle is about 2^{28}.

2) Contention-based attacks: Collision on the target entry so that the old record is evicted is a necessary condition for constructing contention-based attacks. In this case, an attacker learns about the execution of the target branch instructions by sensing whether contention exists. However, TAGE-SC-L predictor has a default base branch predictor. Even if tagged tables cannot give a prediction direction due to contention, different branches typically use and update the default predictor—rather than evicting each other's entries. Furthermore, the default branch predictor is physically isolated and flushed upon context switches, the attacker cannot control the state of the default predictor to perform a contention-based attack. Therefore, HyBP can defeat contention-based attacks on PHT.

C. Analysis on the Frequency of Key Changing

For security, the keys need changing before an attacker can complete any attack. As analyzed before, the shortest attack time for the hybrid protected predictor is at least 2^{27} accesses. Then the value is set as counter threshold by OS. This is longer than the default Linux thread time slice of 4 milliseconds (or 2^{24} 4GHz CPU cycles). Therefore, changing the keys upon context switch is convenient and can satisfy security requirements. Of course, the system can also change the keys at a preset frequency regardless of context switching. We count both speculative and non-speculative BPU accesses using a dedicated counter. When the counter exceeds the threshold, a keys changing request is sent to the randomized index tables. Once the refresh is triggered, the counter is reset to zero. Considering within a default Linux thread time slice, neither BTB nor PHT can be compromised, thus BTB and PHT can share the random tables without security degradation.

When target victim branches increases from 1 to 16, the BPU accesses drops from 2^{28} to 2^{24}. It is close to the interval of OS scheduling. First, when the victim branches were less than 16, HyBP can protect it without performance loss. But if victim branches were more than 16, frequent refreshes will not pose a security risk, but might cause possible performance loss. Considering that more than 16 branches dependent on secret in a real victim application is rare (e.g. square-and-multiply exponentiation function in RSA), a feasible solution is to make code check and scheduling by the compiler, thus avoiding the code segments containing more than 16 secret-dependent branches.

D. Attack & Defense Experiments

To evaluate the effectiveness of our mechanism, we conduct experiments with proof-of-concept (PoC) attacks for BTB and PHT respectively on our FPGA-based processor prototype based on the open-source Berkeley Out-of-Order RISC-V processor (configuration is introduced in Table IV). The PoC is open-sourced at *https://anonymous.4open.science/r/PoC-codes-Branch-Predictor-Attacks-DFBD/*.

In our experiment, we repeat the attack 10000 iterations. A successful attack means that the victim branch jumps to the trained direction more than 90 times per iteration. For the baseline processor without any defense mechanism, the accuracy of training BTB and PHT is 96.5% and 97.2%, respectively. With hybrid protection mechanism, the accuracy of training both BTB and PHT decreases to less than 1%.[1] In summary, our mechanism introduces effective protection against these attacks.

VII. Evaluation

A. Methodology

Since our FPGA processor prototype does not yet support SMT or hierarchical branch predictors, we modeled an out-of-order processor using the cycle-level Gem5 simulator [20]. This processor can support both single-threaded core and SMT core modes. Both cores are modeled after the latest Intel Sunny Cove core [35] as shown in Table IV. We experimented on the branch predictors with a three-level BTB as in AMD Zen2 [21] and TAGE-SC-L [31] predictor. The modifications of hierarchy BTB and TAGE-SC-L predictor with *hybrid protection* mechanism are shown in Figure 3. We have implemented two other defense mechanisms (Flush and Physical isolation) for comparison.

We adopt SPEC CPU2017 benchmarks [36] with reference input size for performance evaluation. For the SMT experiments, pair-wise combinations are selected according to the standard methodology [37], [38]. These combinations fairly represent the spectrum of performance and branch prediction characteristics. To make a fair comparison of our policy, we distinguish between three types of workloads: H-ILP, L-ILP, and MIX. H-ILP workloads contain only high ILP thread, L-ILP workloads contain only low ILP threads, and MIX workloads contain a mixture of both. High-ILP benchmarks include cactuBSSN, imagick, wrf, namd and exchange2. Low-ILP benchmarks include bwaves, cam4, lbm, mcf, xalancbmk and xz. The resulting combination is shown in Table V. The simulator is warmed up for one billion instructions, and then run another billion instructions in the cycle-accurate mode. Two separate metrics are used respectively for the raw execution performance and the execution fairness [37]. For performance, we measure IPC throughput, the sum of the IPC values of all running threads, as it measures how effectively resources are being used. For fairness, we use the Hmean metric proposed in [37], [39]. Hmean measures the harmonic mean of the IPC speedup (or slowdown) of each separate thread, exposing artificial throughput improvements achieved by providing resources to the faster threads.

Table IV: OoO Processor Core Configurations.

Parameter	**Configurations**	
	FPGA prototype	**Gem5 simulation**
ISA	RISC-V	ARM
Frequency	2*GHz* (FPGA @ 50*MHz*)	2.5*GHz*
Processor type	4-decode,4-issue,4-commit	8-decode,8-issue,8-commit
Pipeline depth	10 stages	19 stages, fetch 4 cycles
ROB/LDQ/STQ	64/16/16 entries	352/128/72 entries
Issue Queue	20/16/10 (mem/int/flt)	120
BTB	256 × 2-way	1024 × 4-way, 4 cycles
PHT	TAGE: 33 KB	TAGE-SC-L: 66.6KB
ITLB/DTLB	8/8 entries	64/64 entries
L1 ICache	32KB, 8-way, 64B line	32KB, 4-way, 64B line
L1 DCache	32KB, 8-way, 64B line	48KB, 4-way, 64B line
L2 Cache	1MB, 16-way, 64B line	512KB, 16-way,64B line
L3 Cache	None	4MB, 32-way, 64B line

Table V: Benchmark sets.

Type	**Workloads**	**Workloads**
H-ILP	mix1: cactuBSSN_r+imagick_r	mix2: wrf_r+namd_r
H-ILP	mix3: fotonik3d_r+exchange_r	mix4: wrf_r+cactuBSSN_r
MIX	mix5: imagick_r+xz_r	mix6: imagick_r+bwaves_r
MIX	mix7: wrf_r+mcf_r	mix8: namd_r+roms_r
L-ILP	mix9: xz_r+cam4_r	mix10: cam4_r+xalancbmk_r
L-ILP	mix11: lbm_r+bwaves_r	mix12: cam4_r+bwaves_r

[1]The 1% apparently successful attacks stem from the limitations of the RISC-V experimental platform and software noises. For example, we determine the success of attack by observing Flush+Reload cache side channels. However, flushing a cache line precisely is not supported in RISC-V instruction set, so we flush the whole cache with large arrays. This presents false positive measurement noises on successful attacks. However, an adversary cannot exploit these noises to construct attacks.

Experimental setups with different defense mechanisms are named as follows:

Baseline: The original OoO processor using branch predictor without any defense mechanism.

HyBP: Both BTB and PHT are equipped with our hybrid protection mechanism.

Flush: Flushing the predictor completely upon context switches or privilege changes.

Partition: Allocating separate branch tables for different threads and/or different privilege levels.

Replication: Replicating last-level BTB for each thread and partition each table among user/privilege.

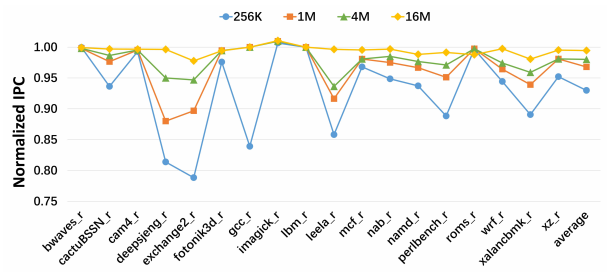


Figure 5: Normalized IPC of *hybrid protection* mechanism on a single-threaded core.

B. Performance Evaluation on Single-threaded Cores

We start with the performance degradation of our proposed hybrid protection mechanism using isolation-randomization (HyBP). Figure 5 shows the performance impact for individual applications under different context switching intervals. We see that some applications have a high sensitivity to branch predictor behavior and the frequency of key changes can result in close to 21% performance degradation under very frequent context switchings. But on average, and taking a moderate interval size of 16M cycles (default context switch length in a typical Linux at 4GHz), the performance cost for protection is less than 0.5% and the performance cost caused by BTB accounted for 74% of the overall performance cost.

We now compare HyBP to two alternatives: flush-based protection (*Flush*) and partition-based physical isolation (*Partition*). Figure 6 shows the resulting average performance degradation under different context switching intervals (from 256K to 16M cycles). We should start at the interval size 16M cycles (default context switch length). We observe that our HyBP's 0.5% performance degradation is much smaller than 6.3% and 5.1% for the other two mechanisms. Also, across the range of context switch intervals, HyBP maintains its performance advantage.

The main source of performance loss for HyBP is context switching: When an application resumes its execution after being swapped out and back again on the baseline, it will benefit from its residual state in the BPU. With all three protection mechanisms, the residual state is either explicitly flushed or inaccessible due to the change of key.

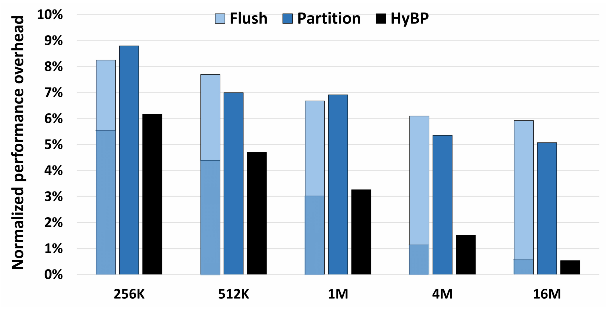


Figure 6: Average performance degradation of three different protections for a single-threaded core under different context switching intervals (The shaded part of the Flush bar represents the performance loss caused by context switch flush).

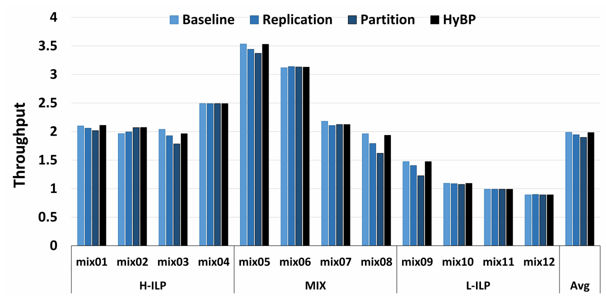
However, the other two mechanisms have extra sources: additional flushes due to privilege level changes (*Flush*) and performance loss due to reduction in available branch predictor resource (*Partition*). We can see that when the period of context switching is 16M cycles or longer, the impact from context switching becomes minimal for all three mechanisms. For the other two alternatives, the performance cost remains high due to their respective causes.

Specifically, for some branch-sensitive test cases such as fotonik3d and xz, the performance loss of *Partition* reaches 18.2% and 19.4% respectively under the interval size of 16M cycles. For other test cases that are sensitive to context switch, especially for deepsjeng, the performance loss of HyBP and *Partition* at the 256K cycles is 14.4% and 10.7%, respectively. When the context switching period increases to 16M cycles, the performance loss of HyBP is reduced to 0.6%–far better than *Partition* (7.9%).

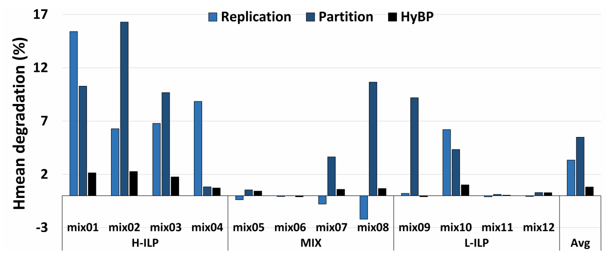
C. Evaluation on SMT Cores

Figure 7 compares performance impacts of physical partitioning and HyBP mechanisms on an SMT core. Here we do not compare with *Flush* mechanism like in a single-threaded core. This is because *Flush* is no longer a solution as it fails to protect against certain attacks in SMT cores. The case for SMT is a bit more complicated than in a single-threaded core: there may be resource conflicts in the baseline system that the protection mechanism can either ameliorate or exacerbate due to chance. And indeed, we find that there are more cases where the protection mechanism actually improves upon the baseline's performance. Overall, we see that HyBP is consistently a high-performance option in Figure 7(a), with a maximum loss of 3.8% compared to 12.6% and 8.9% for physical partitioning and replicating. On average, the impact is 0.2% for HyBP compared to 4.4% for the *Partition*.

In terms of fairness, HyBP also outperforms *Partition* and *Replication* as can be seen in Figure 7(b). Due to the fierce competition for resources, H-ILP workloads are more sensitive to the *Partition* and the performance fluctuations are also more severe than HyBP. And we see that the maximum Hmean degradation of mix02 for *Partition* is nearly 17% compare to 2.3% for HyBP. Even for MIX and L-ILP types of workloads where resource competition is less, HyBP still shows much less Hmean degradation (less than 1% in all cases) compared to *Partition* (up to 11% and an average of 3.6%). Overall, HyBP is clearly more practical than *Partition* on SMT cores.



(a) IPC throughput



(b) Hmean fairness degradation

Figure 7: Throughput and Hmean fairness degradation of four isolation mechanisms on an SMT core with default context switch interval length in a typical Linux at 4GHz.

D. Hardware Cost Estimation

The hardware cost of hybrid protection mechanism consists of three parts: replication of L0 and L1 BTB, base predictor, random tables and encryption modules. We will use a 2-way SMT as an example in calculating area cost.

1) The total storage need for the replicated L0 and L1 BTB and base direction predictor is 16.3 KB.
2) The random tables take total of 5 KB. BTB and TAGE predictor share the same random tables and Each random table takes 1.25 KB. A SMT-2 core requires four tables for (thread, privilege) combinations.
3) Finally, QARMA-64 cipher has an area of $1238.1um^2$ in a state-of-the-art FinFet 7 nm technology [32], which is roughly equivalent to 1.4 KB storage.

All told, the area cost is roughly equivalent to 22.7 KB or 21.1% of the cost of the branch predictor.

Figure 8 explores the area/performance tradeoffs compared to Replication and HyBP. We evaluate a variety of design points of Replication on a scaled-up branch predictor. As the storage overhead of the branch predictor increases in the range of 0 to 300%, the performance loss caused by Replication gradually decreases. At 0 point, each hardware

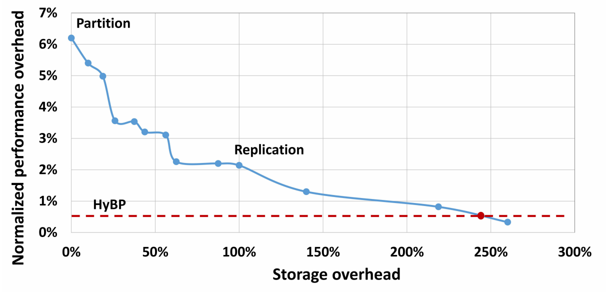


Figure 8: The impact of the replication mechanism on performance as the branch predictor scales up on SMT-2 cores.

thread uses the same branch predictor resources as Partition. When the storage overhead increases to 100%, the branch predictor portion that each hardware thread uses are equivalent to the Replication. Until the storage overhead reaches about 240%, the performance overhead is roughly equivalent to that of HyBP (perf loss 0.5%, overhead 21.1%).

E. Sensitivity Analysis on the Size of Keys Tables

Table VI: Performance overhead under different randomized index keys table sizes

		Entries per keys table				
		1K	**2K**	**4K**	**16K**	**32K**
Context switching interval (cycle)	**4M**	1.4%	1.5%	1.7%	1.8%	1.9%
	16M	0.5%	0.5%	0.6%	0.7%	0.9%

When context switches, the current thread's keys in the randomized index keys table are updated. It should be noted that refreshing randomized index tables takes hundreds of cycles. During this phase, the branch instruction uses the result trained by the old key, causing performance loss due to mispredictions. Table VI shows the performance overhead under different randomized index keys table sizes. With the increase of keys table, the performance overhead of HyBP increase accordingly. When the keys tables increase from 1K to 32K, the performance overhead increases from 0.5% to 0.9% with typical context switch interval length. At this time, HyBP and BRB have similar performance, but the storage overhead is half of BRB.

F. Comparisons

There are many ways to design a secure branch predictor. HyBP achieves a good balance between security, performance, and implementation cost. In the following, we discuss the earlier Table I in detail.

Flush, physical partitioning, and disabling SMT bring performance degradation/throughput reduction of 5.1%, 6.3%, and 18% respectively. To put such performance degradation in perspectives, the performance gain from using the latest TAGE-SC-L over the decades-old tournament predictor in our setup is just 5.4%. Thus, for general-purpose processors, even performance degradation/improvement under 10% is crucial. One should not settle with the 5% $\sim$ 6% performance degradation and stop innovating new ideas. In contrast, HyBP *only incur a performance degradation of 0.5% with a very modest area overhead of 21.1%*. We claim that it is a major advance of the state-of-the-art.

Specifically, Flush has a marginal hardware cost. But it fails to protect against certain attacks in SMT cores since flushing only happens during context or privilege switches. The performance overhead of the Flush mechanism is due to context switch flushing and privilege change flushing which are used to defend against cross-privilege attacks and cross-thread attacks respectively. As shown in Figure 6, when the context switching interval is below 512K cycles, the privilege change flushing is the majority (>57%); otherwise, context switching is dominant.

For physical isolation, we consider two methods. 1) Partition: If we keep the size unchanged and divide the branch predictor evenly, the *performance loss of partition increases by about 5.1% on average (and up to 19.4%)* compared to our hybrid protection mechanism. It is similar to throwing away decades of performance improvements from advances in branch predictors. 2) Replication: Assigning separate tables to all thread privilege level combinations is also impractical. Taking the SMT-2 processor as an example, assuming that only user and kernel privileges are considered. If we just replicate for each thread and partition each table among user/kernel so each privilege level can only use 50% table, *the performance loss is 2.1%, but the storage overhead is 100%* Alternatively, we scale up the branch predictor to make its performance equal to HyBP, but the storage overhead reaches 240%—clearly far worse than HyBP (perf loss 0.5%, overhead 21.1%).

BRB [12] is a state-of-art mitigation mechanism that can be considered as a replication mechanism since it maintains a small checkpoint of branch predictor states for each context upon context switches and restores it once the context becomes active. The hardware cost of one BRB checkpoint is about 6.6KB (BTB: 2.6KB, bimodal:1KB, and TAGE predictors: 3KB). The storage overhead grows with more contexts. Even with the recommended three checkpoints per hardware thread, the storage overhead is more than twice that of HyBP.

VIII. Related Work

Countermeasures against the vulnerabilities in branch predictor can be classified into four categories.

Reducing information leakage through the side channel. For instance, branches that carry sensitive information can be transformed into safe instructions that do not leave a mark in the branch predictors [7]. Limiting performance counter usage can reduce the information obtained by the attacker [40]. InvisiSpec [41], Conditional Speculation [42], CleanupSpec [43], and STT [44] prevent speculative execution from generating visible microarchitectural state.

Flushing the predictor tables to contain randomized new results. Performing this by software during context switch can bring non-trivial overhead [9]. Such expensive operations can be limited to only the sensitive processes [45]. The impact on performance and prediction accuracy of flushing predictor table in hardware has been studied [46], [47]. The longer the context switch interval, the smaller the impacts. These observations are consistent with ours.

Using *dedicated hardware* as a general approach to isolate from different processes. Sensitive applications in SGX can be provided with their own branch predictor tables [2]. Earlier work on performance improvement considered saving and restoring compressed branch prediction information [48] or providing thread-private branch predictors on SMT processors [49]. BRB is a proposal to retain partial predictor state in on-chip SRAM banks per context and select a correct entry for the active context when a context switch occurs [12].

Randomizing index and content of branch predictor has been proposed. Samsung Exynos has implemented content encryption via simple substitution cipher in branch-target buffers and return address stack [13], but it only protects against some Spectre variants (*e.g.*, Spectre V2 and Spectre RSB). Lee et al. [14] and Zhao et al. [15] propose to randomize the index of branch predictor to mitigate branch predictor side-channels using low-latency cipher. However, they use the LLBC proposed by CEASER [16]. LLBC has been proven to be linear and vulnerable to cryptanalytic attacks [17], [18].

IX. Conclusion

This paper proposes HyBP, a practical hybrid protection and effective mechanism for building secure branch predictors. HyBP protects the smaller tables with physical isolation based on (thread, privilege) combination; and protect the large tables with randomization. Surprisingly, the physical isolation also significantly enhances the security of the last-level tables by naturally filtering out accesses, reducing the information flow to these bigger tables. As a result, key changes can happen less frequently and be performed conveniently at context switches. Overall, our design incurs a performance penalty of 0.5% compared to 5.1% of physical isolation under the default context switching interval in Linux.

Acknowledgment

We sincerely thank the anonymous reviewers for their insightful comments and suggestions. This work is supported by the Strategic Priority Research Program of Chinese Academy of Sciences under grant No. XDC02010200 and the National Science Fund for Distinguished Young Scholars under grant No. 62125208. Rui Hou is the corresponding author.

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 *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1–19.

[2] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, "BranchScope: A new side-channel attack on directional branch predictor," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, 2018, pp. 693–707.

[3] T. Huo, X. Meng, W. Wang, C. Hao, P. Zhao, J. Zhai, and M. Li, "Bluethunder: A 2-level directional predictor based side-channel attack against sgx," in *IACR Transactions on Cryptographic Hardware and Embedded Systems*. Springer, 2019, pp. 321–347.

[4] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Covert channels through branch predictors: a feasibility study," in *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 2015, pp. 1–8.

[5] O. Aciiçmez, Ç. K. Koç, and J.-P. Seifert, "On the power of simple branch prediction analysis," in *Proceedings of the 2nd ACM symposium on Information, computer and communications security*. ACM, 2007, pp. 312–320.

[6] T. Zhang, K. Koltermann, and D. Evtyushkin, "Exploring branch predictors for constructing transient execution trojans," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 667–682.

[7] G. Agosta, L. Breveglieri, G. Pelosi, and I. Koren, "Countermeasures against branch target buffer attacks," in *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2007)*. IEEE, 2007, pp. 75–79.

[8] Intel, "Speculative execution side channel mitigations," https://software.intel.com/security-software-guidance/api-app/sites/default/files/336996-Speculative-Execution-Side-Channel-Mitigations.pdf, 2017.

[9] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Understanding and mitigating covert channels through branch predictors," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, 2016.

[10] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside {SGX} enclaves with branch shadowing," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 557–574.

[11] M. Taram, A. Venkat, and D. Tullsen, "Context-sensitive fencing: Securing speculative execution via microcode customization," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2019, pp. 395–410.

[12] I. Vougioukas, N. Nikoleris, A. Sandberg, S. Diestelhorst, B. M. Al-Hashimi, and G. V. Merrett, "Brb: Mitigating branch predictor side-channels." in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 466–477.

[13] B. Grayson, J. Rupley, G. Z. Zuraski, E. Quinnell, D. A. Jiménez, T. Nakra, P. Kitchin, R. Hensley, E. Brekelbaum, V. Sinha, and A. Ghiya, "Evolution of the samsung exynos cpu microarchitecture," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 40–51.

[14] J. Lee, Y. Ishii, and D. Sunwoo, "Securing branch predictors with two-level encryption," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 17, no. 3, pp. 1–25, 2020.

[15] 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 *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 1–6.

[16] M. K. Qureshi, "Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping," *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 775–787, 2018.

[17] A. Purnal, L. Giner, D. Gruß, and I. Verbauwhede, "Systematic analysis of randomization-based protected cache architectures," in *42th IEEE Symposium on Security and Privacy*, 2021.

[18] R. Bodduna, V. Ganesan, P. SLPSK, K. Veezhinathan, and C. Rebeiro, "Brutus: Refuting the security claims of the cache timing randomization countermeasure proposed in ceaser," *IEEE Computer Architecture Letters*, pp. 9–12, 2020.

[19] J. Borghoff, A. Canteaut, T. Güneysu, E. B. Kavun, M. Knezevic, L. R. Knudsen, G. Leander, V. Nikov, C. Paar, C. Rechberger, P. Rombouts, S. S. Thomsen, and T. Yalçın, "Prince: A low-latency block cipher for pervasive computing applications (asiacrypt'12)," in *Proceedings of the 18th International Conference on The Theory and Application of Cryptology and Information Security*, 2012, p. 208–225.

[20] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, and S. Sardashti, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[21] D. Suggs, M. Subramony, and D. Bouvier, "The AMD "Zen 2" processor," *IEEE Micro*, pp. 45–52, 2020.

[22] ANANDTECH, "Arm's new Cortex-A78 and Cortex-X1 microarchitectures: An efficiency and performance divergence," https://www.anandtech.com/show/15813/arm-cortex-a78-cortex-x1-cpu-ip-diverging/3, 2020.

[23] IBM, *POWER9 Processor User's Manual. Version 2.1*, 2019.

[24] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "ScatterCache: Thwarting cache attacks via cache set randomization," in *28th USENIX Security Symposium (USENIX Security 19)*, Aug. 2019, pp. 675–692.

[25] F. Liu and R. B. Lee, "Random fill cache architecture," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 203–215.

[26] 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.

[27] M. K. Qureshi, "New attacks and defense for encrypted-address cache," in *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19)*, 2019, p. 360–371.

[28] B. Jo, Van, P. Frank, and S. Raoul, "Sgx-step: A practical attack framework for precise enclave execution control," in *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, 2017, pp. 4:1–4:6.

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

[30] E. J. Ojogbo, M. Thottethodi, and T. N. Vijaykumar, "Secure automatic bounds checking: Prevention is simpler than cure," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization (CGO 2020)*, 2020, p. 43–55.

[31] A. Seznec, "TAGE-SC-L Branch Predictors Again," in *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, Seoul, South Korea, Jun. 2016.

[32] R. Avanzi, "The QARMA block cipher family. almost MDS matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency s-boxes," *IACR Trans. Symmetric Cryptol.*, vol. 2017, no. 1, pp. 4–44, 2017.

[33] Intel, "Digital random number generator (drng) software implementation guide," https://software.intel.com/en-us/articles/intel-digital-random-number-generator-drng-software-implementation-guide, May 2014.

[34] J. S. Liberty, A. Barrera, D. W. Boerstler, T. B. Chadwick, S. R. Cottier, H. P. Hofstee, J. A. Rosser, and M. L. Tsai, "True hardware random number generation implemented in the 32-nm soi power7+ processor," *IBM Journal of Research and Development*, vol. 57, no. 6, pp. 4:1–4:7, 2013.

[35] WikiChip, "Sunny Cove - microarchitectures - intel," 2019. [Online]. Available: https://en.wikichip.org/wiki/intel/microarchitectures/sunny_cove

[36] J. Bucek, K.-D. Lange, and J. v. Kistowski, "SPEC CPU2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '18, 2018, p. 41–42.

[37] F. J. Cazorla, A. Ramirez, M. Valero, and E. Fernandez, "Dynamically controlled resource allocation in smt processors," in *37th International Symposium on Microarchitecture (MICRO-37'04)*, 2004, pp. 171–182.

[38] J. Vera, F. J. Cazorla, A. Pajuelo, O. J. Santana, E. Fernandez, and M. Valero, "Fame: Fairly measuring multithreaded architectures," in *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, 2007, pp. 305–316.

[39] Kun Luo, J. Gummaraju, and M. Franklin, "Balancing thoughput and fairness in smt processors," in *2001 IEEE International Symposium on Performance Analysis of Systems and Software. ISPASS.*, 2001, pp. 164–171.

[40] P. Guide, "Intel® 64 and ia-32 architectures software developer's manual," *Volume 3B: System programming Guide, Part*, vol. 2, 2011.

[41] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "Invisispec: Making speculative execution invisible in the cache hierarchy," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 428–441.

[42] 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 *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 264–276.

[43] G. Saileshwar and M. K. Qureshi, "CleanupSpec: An "Undo" Approach to Safe Speculation," *international symposium on microarchitecture*, pp. 73–86, 2019.

[44] 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 *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2019, pp. 954–968.

[45] W.-M. Hu, "Lattice scheduling and covert channels," in *Proceedings 1992 IEEE Computer Society Symposium on Research in Security and Privacy*. IEEE, 1992, pp. 52–61.

[46] M. Evers, P.-Y. Chang, and Y. N. Patt, "Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches," in *ACM SIGARCH Computer Architecture News*, vol. 24, no. 2. ACM, 1996, pp. 3–11.

[47] S. Pasricha and A. Veidenbaum, "Improving branch prediction accuracy in embedded processors in the presence of context switches," in *Proceedings 21st International Conference on Computer Design*. IEEE, 2003, pp. 526–531.

[48] A. S. Dhodapkar and J. E. Smith, "Saving and restoring implementation contexts with co-designed virtual machines," in *Workshop on Complexity-Effective Design*. Citeseer, 2001.

[49] M. Ramsay, C. Feucht, and M. H. Lipasti, "Exploring efficient smt branch predictor design," in *Workshop on Complexity-Effective Design, in conjunction with ISCA*, vol. 26, 2003.