Exploring Transient Execution Vulnerabilities, Side-Channel Attacks, and Defenses

Tao Zhang

Beijing, Beijing, China

Bachelor of Engineering, North China University of Technology, 2012
Master of Science, Central Michigan University, 2014

A Dissertation presented to the Graduate Faculty of
The College of William and Mary in Virginia in Candidacy for the Degree of
Doctor of Philosophy

Department of Computer Science

The College of William and Mary in Virginia
May 2024

© Copyright by Tao Zhang 2024

APPROVAL PAGE

This Dissertation is submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

Tao Zhang

Approved by the Committee, May 2024

Committee Chair
Dmitry Evtyushkin, Assistant Professor, Computer Science
College of William & Mary

Bin Ren, Associate Professor, Computer Science
College of William & Mary

Robert Michael Lewis, Associate Professor, Computer Science
College of William & Mary

Stephen Herwig, Assistant Professor, Computer Science
College of William & Mary

Dr. Antonio Gomez Iglesias
Intel Corporation

ABSTRACT

Modern microprocessors utilize branch prediction and speculative execution to enhance instruction throughput. Instead of stalling the pipeline and waiting for branch targets to be computed, the CPU consults branch predictors for a possible destination and performs speculative execution. These microarchitectural techniques improve the efficiency of instruction pipelining and out-of-order execution, enabling higher performance and better resource utilization. Despite their widespread adoption, the potential security implications of branch misprediction and transient execution have not drawn much attention until recently.

Around early 2018, the discovery of Spectre attacks exposed critical vulnerabilities in CPUs, undermining both software and hardware isolation and confidentiality. These attacks exploit the side effects of speculative execution stemming from branch predictions. By manipulating branch predictors to generate incorrect predictions, an attacker can trigger speculative execution to bypass bound checks or operate on arbitrary memory space. Consequently, such exploits can access sensitive data during speculative execution and then exfiltrate the information through various microarchitectural side channels. Spectre and its variants pose a significant security threat that is challenging to mitigate, and existing defenses often come with substantial performance overheads. This dissertation tackles the threat from two perspectives. We first enhance the understanding of exploitable hardware primitives by introducing new transient trojan attacks. Second, we propose secure microarchitecture designs without compromising performance.

We first challenge the perception that the triggers and effects of transient execution attacks are fully understood and that the existing protections leave no room for any attack to occur. We present transient trojans, software modules that conceal malicious activity within transient execution mode. These trojans appear entirely benign, pass static and dynamic analysis checks, but reveal sensitive data when triggered. To construct these trojans, we conducted a comprehensive analysis of the current attack surface in light of recommended mitigation techniques. We uncovered new exploitation techniques through reverse-engineering branch predictors in a selection of recent x86_64 processors. Leveraging these findings, we design three types of transient trojans, showcasing their ability to evade detection and their effectiveness. Second, we present the secret token branch predictor unit (STBPU), a secure BPU design to defend against collision-based speculative execution attacks and BPU side channels with minimal performance impact. Securing branch predictors is challenging, as techniques like partitioning or flushing the BPU only partially mitigate collision-based exploits. Moreover, such mitigations compromise branch prediction accuracy, leading to overall CPU performance degradation. STBPU resolves these challenges by customizing BPU data representations for each software entity that requires isolation. Furthermore, STBPU monitors related hardware events and preemptively adjusts BPU data representations.

TABLE OF CONTENTS

Acknowledgments **v**

Dedication **vi**

List of Tables **vii**

List of Figures **viii**

1 Introduction **2**

1.1 Introduction 2

1.1.1 Exploring Branch Predictors for Constructing Transient Execution Trojans 3

1.1.1.1 Problem Statement 3

1.1.1.2 Contributions 4

1.1.2 STBPU: A Reasonably Safe Branch Predictor Unit 5

1.1.2.1 Problem Statement 5

1.1.2.2 Contributions 5

1.2 Organization 7

2 Background & Motivation **8**

2.1 Branch Prediction and BPU structures 8

2.2 Transient Execution Attacks 12

2.3 BPU Centered Attacks 13

2.4 Existing Protections of BPU Transient Execution Attacks 15

2.4.1 Retpoline Sequences 15
2.4.2 System-wide Microcode-Based Protections 16
2.5 Transient Trojan Motivation and Current Attack Surface 16
2.6 STBPU's Motivation and Protection Scope 17

3 Branch Predictor & Transient Execution Trojans 21
3.1 Introduction 21
3.1.1 Threat Model and Assumptions 23
3.2 Transient Execution Trojans 24
3.2.1 Branch Target Prediction Mechanisms 24
3.2.1.1 Addressing Modes for Indirect Branch Prediction . . . 27
3.2.1.2 Selecting Branch Type for BTB Poisoning 29
3.2.1.3 Finding Branch Collisions 30
3.2.1.4 Predictor selector mechanism 31
3.2.2 Distant Collision Trojans 33
3.2.2.1 Trojan example utilizing a system call 34
3.2.3 Portable Trojans 37
3.2.3.1 Early Front-end Branch Collisions 37
3.2.3.2 Constructing a Portable Trojan 41
3.2.3.3 Dispersing Gadgets to Avoid Detection 43
3.2.4 Skipping Branch Trojans 44
3.2.4.1 Skipping indirect branches 44
3.2.4.2 Skipping based transient execution attack 45
3.3 Improving Trojan Activation Rate 46
3.4 Detecting Collisions in Existing Binaries 48
3.5 Countermeasures 50
3.6 Related Work 51

3.7 Conclusions 52

4 STBPU: A Reasonably Safe Branch Predictor Unit 53

4.1 Introduction 53

4.2 Threat Model 55

4.3 STBPU Design 56

4.3.1 ST re-randomization 57

4.3.2 Hardware Mechanisms and Interfaces 58

4.4 Implementation 60

4.4.1 Automation of Finding Remapping Functions 62

4.4.2 Optimization and Remapping Selection 64

4.5 Security Analysis 65

4.5.1 Analysis of Branch Predictor Attacks under STBPU 65

4.5.1.1 Target Injection Attacks 66

4.5.1.2 Reuse-based Attacks 67

4.5.1.3 Same Address Space Attacks 68

4.5.1.4 Eviction-based Attacks 68

4.5.1.5 Re-randomization Thresholds for Baseline Model . . . 70

4.5.1.6 Denial-of-Service Attack on STBPU 70

4.6 Evaluation of STBPU Design 71

4.6.1 Re-randomization Threshold 73

4.6.2 STBPU Performance Evaluation 73

4.6.2.1 Prediction Accuracy with real branch trace 73

4.6.2.2 Cycle Accurate Evaluation using gem5 75

4.6.2.3 Aggressive ST Re-randomization and Performance . . 76

4.7 Related Work 77

4.7.1 Existing Microcode-based Mitigations 77

4.7.2 Other Defense Directions against Spectre Attacks 78
4.7.3 Existing Secure Designs of BPU 79
4.8 Conclusions & Accessibility 79

5 Conclusion **84**
5.1 Main Contributions 84
5.2 Implications and Future Directions 85

Bibliography **87**

Vita **109**

ACKNOWLEDGMENTS

I sincerely appreciate my advisor, Dmitry Evtyushkin, for his diligent mentoring. Over the past year, he has taught me many important aspects of being a true researcher, encompassing not just skills and expertise but also curiosity, integrity, and persistence.

I am grateful to the members of my committee—Bin Ren, Robert Michael Lewis, Stephen Herwig, and Antonio Gomez Iglesias—for their generous support and insightful feedback. I also thank Pawan Kumar Gupta, Daniel Sneddon, and other Intel experts for their suggestions and technical advice during my internship. I also thank my master's advisor, Qi Liao, who encouraged me to embark on my research journey.

I would also like to acknowledge the faculty and staff at the Department of Computer Science, especially Vanessa Godwin, for their professionalism and care.

Lastly, I am indebted to my family and friends, whose unwavering support keeps me under their wings so I can explore the world and enjoy life freely.

To my grandpa Yongqi Zhang, my wife Chuqiu, and all my family.

LIST OF TABLES

2.1 Attack surface classification for BPU collision-based attacks by event and adversarial effect types 13

3.1 Misprediction rate observed in two different patterns composed by varying the BHB context. H represents hit, and M represents misprediction 29

4.1 I/O bits for baseline and STBPU functions 59

4.2 Parameters used in STBPU analysis 66

4.3 Parameters used in gem5 simulation 72

LIST OF FIGURES

2.1 A branch predictor components and workflow 10

2.2 Transient execution attack surface 17

2.3 STBPU protection region v.s. the others 18

3.1 Branch target prediction mechanism combining direct and indirect branch prediction logic. Functions $f1-3$ are bit compression functions; $f4$ is bit matching function. Mechanisms used for trojan construction are highlighted red 27

3.2 Collision detection experiment setup 31

3.3 Misprediction patterns demonstrating the competition between the two addressing modes and an FSM matching this behavior 32

3.4 Transient trojan based on `open()` system call 36

3.5 Branch collision patterns within the same cache line on Haswell and Skylake CPUs 39

3.6 Demonstration of the two collisions types 40

3.7 Portable transient trojan example 43

3.8 Dispersing a transient gadget to avoid gadget detection tools. Solid arrows indicate transient execution flow 44

3.9 Transient trojan based on branch skipping 46

3.10 Genetic and randomization optimizer comparison 48

3.11 Analysis of branch collisions in existing binaries 50

4.1 BPU with STBPU components highlighted 59

4.2 R_1 remapping function construction 64
4.3 Overall branch prediction accuracy: STBPU against other secure BPU models 71
4.4 STBPU single workload evaluation in gem5 81
4.5 Gem5 Multi-workload (SMT) Evaluation of STBPU 82
4.6 Effects on performance when using more aggressive re-randomization thresholds with the TAGE_SC_L_64KB BPU, result are averaged from 42 combinations of SPEC CPU 2017 workload pairs. The X-axis represents the r parameter. 83

Exploring Transient Execution Vulnerabilities, Side-Channel Attacks, and Defenses

Chapter 1

Introduction

1.1 Introduction

The branch predictor unit (BPU) plays a pivotal role in the modern CPU microarchitecture. From conditional branch to indirect jump, branch instructions have various forms, yet they all sit in the critical link between the instruction fetching and execution. It is well known that fully resolving a branch is a slow process, and CPU issuing stalls will impede overall machine performance [17, 17, 46, 135]. Thus, rather than waiting for the branch resolution, the modern CPU employs BPU to guess its outcome using different prediction schemes [161, 129, 68, 98]. Every branch destination predicted from BPU then facilitates the speculative execution on the potential path. As a result, BPU becomes an essential component at the front-end, guarding the performance and efficiency of CPU deep pipelining.

Despite many works [161, 129, 131, 133, 132, 130, 68, 98] being proposed to improving BPU performance, the security of BPU has been an overlooked topic. In early 2018, the Spectre attack [79] revealed BPU could be manipulated to derive transient execution on arbitrary locations, including malicious payloads. As a result, attackers can bypass bound checks and detection, gain high privilege, access user secrets, etc. More importantly, the

nature of branch prediction and the fundamental aspects of using prediction to trigger speculative executions makes securing BPU and CPU front-end a challenging task. So far, no state of the arts can mitigate all attacks nor operate without heavily affecting the performance. On the other hand, more exploits [33, 95, 76, 81, 128] have been discovered, escalating the threat on commodity machines.

This dissertation focuses on addressing the BPU security and the related performance challenges against microarchitectural attacks in two folds: 1) reverse-engineering modern CPU branch predictors for hidden vulnerabilities and 2) proposing secure BPU protection mechanisms while minimizing the associated overhead in performance.

1.1.1 Exploring Branch Predictors for Constructing Transient Execution Trojans

1.1.1.1 Problem Statement

As noted above, although the security community has been proposing protections since the debut of the Spectre attack, we believe that the triggers and effects of transient execution attacks have not been fully understood. Consequently, the de facto solutions, while being recommended, leave much room for advanced exploits. In particular, hardware manufacturers such as Intel responded to transient execution attacks with microcode updates. The most widely deployed mitigation comes with multiple techniques, eliminating malicious branch poisoning by limiting or disabling BPU lookup. Throttling branch prediction could result in high performance overhead. Thus, the mitigation only takes action at specific scenarios to reduce the cost. However, the validity of its protection region reminds unknown and needs to unfold. This work explores hidden BPU behaviors to uncover exploration mechanisms and constructs advanced attacks to break the existing protection model provided by the widely-used microcode BPU protections.

1.1.1.2 Contributions

We reverse-engineer several branch predictors from modern CPUs, including Intel Haswell (i7-4800MQ), Skylake (i7-6700K), Kaby Lake (i7-8550U), and AMD Ryzen (1950X) for hidden branch prediction mechanisms. Utilizing the uncovered schemes, we present the transient (execution) trojans, a new type of practical attack that conceals the malicious activity within transient execution mode and bypass existing protection and detection. In summary, our research on exploring branch predictors for constructing transient execution trojans makes the following contributions:

- Our reverse-engineering study uncovers three hidden indirect branch prediction mechanisms and anomalies, allowing us to tailor different transient execution trojans.
- We uncover a new branch instruction collision mechanism based on early BPU accesses, allowing attackers to construct small and portable trojans that can avoid being detected by current techniques based on code analysis.
- To improve transient trojans' stealthiness and effectiveness, we propose a technique to disperse transient gadgets.
- We analyze the static prediction mechanism of indirect branch skipping. We demonstrate that such a mechanism is vulnerable to bypassing existing gadget detection techniques and constructing trojans.
- We apply analysis to a wide range of binaries and demonstrate a high prevalence of potentially dangerous branch collisions. Attackers can exploit such naturally occurring collisions for transient trojans construction and obfuscation.
- We provide insights on protection techniques and suggest approaches to remove the threat from uncontrolled transient execution.

1.1.2 STBPU: A Reasonably Safe Branch Predictor Unit

1.1.2.1 Problem Statement

As mentioned before, BPU and BPU research mainly focus on performance. Even under the recent microarchitectural threats such as Spectre [79], the security community tends to mitigate the vulnerability from other related components such as designing secure caches [152, 74, 154, 118, 37, 147, 90] and memory buses [126, 12, 87]. This is because the critical role BPU plays for CPU frontend acceleration cannot tolerate any reduction in performance from security mechanisms. However, the aforementioned indirect solutions, plus a handful of attempts of direct approach on BPU [85, 52, 149], cannot fully address the BPU vulnerabilities. For instance, recent μop caches disclosure primitive [123] can leverage BPU for more powerful transient execution attacks without any implicit or explicit data access, voiding their security guarantee of the state of the art protections such as STT [162] and context-based fencing [140]. Such incompleteness in protection calls for a new safe branch prediction mechanisms that can combine strong security guarantees with low performance overhead. In this work, we propose STBPU, a safe BPU design that is immune to collision-based attacks and BPU side channels.

1.1.2.2 Contributions

We present a secret token branch predictor unit (STBPU) design to protect branch predictors against collision-based side channel and speculative execution attacks. STBPU prevents controlled branch collisions which can lead to unsafe branch mistraining and malicious speculative execution and side-channel. STBPU provides unique branch representation for each software entity in the form of address mappings and encrypting branch lookup information in BPU. Specifically, each software entity is associated with a unique, randomly-generated secret token (ST) that customizes its data representations. In addition, STBPU re-randomizes STs by monitoring runtime hardware events, which stops potential attackers from reverse-engineering the ST tokens.

In summary, our work of STBPU makes the following contributions:

- We propose STBPU, a safe BPU design that provides strong isolation guarantees with low overheads. STBPU protects against speculative execution attacks and eliminates BPU side channels.
- We provide insight on how to adapt the design of STBPU on related front-end components such as defense the recent μop cache attacks, which can bypass state-of-the-art protection mechanisms, e.g., invisible speculation and fencing.
- We provide a detailed analysis of the attack surface and examine STBPU with in-depth security analysis. This allows us to find a balance between isolation, design simplicity, and low performance overhead.
- We build an automated framework to derive lightweight ST-dependent remapping functions, allowing ST processing to fulfill the rigorous strict timing constraint.
- Our security analysis proves secure isolation provided by STBPU against recent fast algorithm attacks and future attacks.
- We evaluate STBPU performance on advanced prediction mechanisms, e.g., TAGE_SC_L and Perceptron, and show low overhead even with extreme security settings.

1.2 Organization

The rest of this dissertation is organized as follows. Chapter 2 provides a comprehensive background on Branch Prediction Units (BPUs) and their role in facilitating speculative execution in modern CPUs. It then explores transient execution vulnerabilities and the attacks centered around BPUs, and reviews the existing mitigations. Furthermore, the chapter highlights the motivation, emphasizing the need of uncovering unknown transient attack vulnerabilities and the importance of secure microarchitectural designs. Chapter 3 explores the construction of transient execution trojans through the manipulation of branch predictors. This chapter details the process of uncovering new vulnerabilities within BPUs and demonstrates the development of novel transient execution attacks. Chapter 4 presents the Secret Token Branch Predictor Unit (STBPU), a novel design aimed at securing BPUs to against collision-based speculative execution attacks and BPU side channels. This chapter elaborates on the design and implementation of STBPU and evaluates its effectiveness in mitigating both known and potential speculative execution attacks without imposing significant performance overheads.

Chapter 5 concludes the dissertation with a summary of the findings and contributions and suggests directions for future work in enhancing processor security against speculative execution attacks.

Chapter 2

Background & Motivation

In this chapter, the first section presents a high-level overview of different BPU substructure principles with a baseline model in recent Intel processors. Then, we provide an overview of BPU design features vulnerable to BPU attacks. Next, we explore the current BPU attack surface with BPU design features exploit

2.1 Branch Prediction and BPU structures

A branch is an instruction that causes any deviation from sequential instruction processing. When CPU front-end encounters branch instructions, the address of next instruction is unknown until branch instruction is fully resolved by the pipeline. To avoid costly pipeline stalls, the front-end relies on BPU to predict most likely execution path and continues processing instructions in speculative mode. If the prediction later determined incorrect, the CPU detects a mispredictions and reverts effects of speculative execution.

Thus, Branch prediction is critical for CPU performance as it allows processes to avoid costly pipeline stalls on branch instructions. Instead of waiting for a branch to fully resolve, the CPU executes further instructions in speculative mode using predictions from the BPU. Mispredictions result in erroneous speculative execution which requires the CPU to restore the state before speculative execution started and re-execute the correct instruction sequence. A good BPU must have low latency to allow speculative execution to begin as

soon as possible and have high accuracy to reduce mispredictions that trigger expensive roll-backs. Since the BPU is situated on the critical path of the CPU front-end, it directly impacts the pipeline performance. Even a slight reduction in BPU accuracy or increase in latency may greatly affect overall CPU speed.

A typical ISA permits the following types of branch instructions. Using x86_64 as an example: i) Direct jumps/calls where target addresses are encoded as an offset from the current instruction pointer and stored as an immediate value. To calculate the target CPU needs to perform an addition. ii) Conditional jumps that are only taken if a certain flag in the flag register is set. The target of this branch is encoded similarly to direct jumps. iii) Indirect jumps/calls where targets are stored in a register or in memory, and can change throughout program execution. iv) Return instructions are a special type of indirect jumps where the target is stored on top of the call stack. In addition, various interrupts and exception-triggering instructions can be viewed as branches. However, they are typically not predicted by the BPU.

Although hardware manufacturers typically keep design details confidential, recent reverse-engineering efforts have unveiled insights of the branch prediction unit (BPU) models used in Intel processors. These findings [42, 79, 164, 44, 93, 83] enable us to construct a baseline model for illustrating and evaluating the STBPU design. The baseline model mirrors the branch predictor, including structure sizes, utilized in the Intel Skylake microarchitecture. STBPU is adaptable to various branch predictor configurations and designs, e.g., [129, 131]. This is possible because STBPU does not interfere with underlying prediction mechanisms. STPBU only alters the branch instruction representation inside different BPU data structures. We validate the efficiency of STBPU by integrating it with several advanced predictors, such as TAGE-SC-L [132] and Perceptron [68], demonstrating that it incurs minimal performance overheads in their STBPU-protected variants (marked with `ST_` prefixes) as elaborated in §4.6.

The BPU consists of the following main components: shift registers such as the global history register (GHR) and branch history buffer (BHB), branch target buffer (BTB),

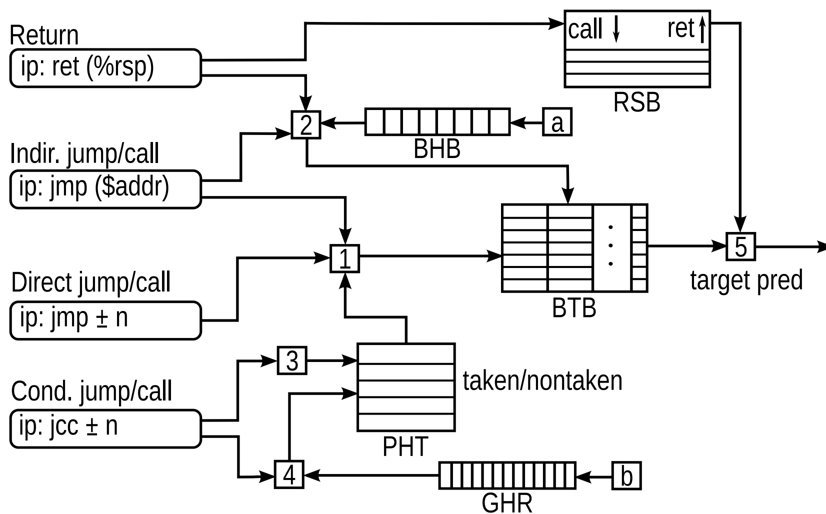


Figure 2.1: A branch predictor components and workflow

pattern history table (PHT), and return stack buffer (RSB). Different structures are used in combination to generate predictions dependent on specific branch types. Figure 2.1 depicts how these structures are utilized during a BPU lookup with highlighted components that are modified by STBPU. The figure also shows several important functions which are referenced later. In addition, two hybrid directional prediction mechanism i.e Tage and Perceptron are also introduced.

Shift registers such as GHR and BHB are used in the BPU as a low-cost way of retaining complex branch history. GHR stores the global history of taken/not-taken branches. For instance, the register shifts and adds 1 or 0 for new taken/not-taken branches, respectively. BHB is used by the indirect branch predictor. Its purpose is to accumulate the branch context. When a branch instruction is executed, its virtual address is folded using XOR and mixed with the current state of BHB [79]. This context is used as part of BTB lookup, enabling BPU to predict the target of an indirect branch when it depends on both branch virtual address and the sequence of previously executed branches.

BTB serves the purpose of caching target addresses of branch instructions. It is implemented as an 8-way, 4096-entry table. Each entry stores a truncated address of the 32 least significant address bits of the target. Function 5 is then utilized to convert a 32-bit entry into a 48-bit

virtual address during prediction by combining 16 higher bits from the branch instruction pointer with 32 lower bits from BTB. While the BTB is used to store targets for all branch types, it has two addressing modes. In mode one, the virtual address of a branch instruction is used to compute an index and tag. In mode two, in addition to virtual address, the BHB is used to perform a lookup. Mode two is only used when predicting indirect branches, and serves as a fall-back mechanism for predicting returns. This addressing enables storing multiple targets for a single indirect branch depending on the context [42, 164, 79].

PHT is a large (16k entry) table consisting of n-bit (e.g. 2-bit) saturating counters; each counter implements a simple finite-state machine with states ranging from strongly non-taken to strongly-taken. This structure is used as a base predictor to predict the direction of conditional branches. Previous studies [62, 22, 164, 44] indicated the presence of a mechanism similar to gshare [161] with two distinct modes of addressing: i) a simple 1-level mode where the virtual memory address of a branch is used to find a PHT entry, and ii) a more complex 2-level mode where the branch virtual memory address is hashed with global history register (GHR), enabling accurate prediction of the branches with complex patterns.

RSB is used to predict return instructions. The RSB is implemented as a fixed size (16-entry) hardware stack [93, 83]. A call instruction pushes a return address on the RSB, and a return instruction pops it. Similar to the BTB, RSB stores only 32 bits of the target. Due to limited capacity, the RSB can underflow. In this case, returns are treated as indirect branches, and the indirect predictor is utilized for prediction.

Hybrid Directional Predictors combine a base directional predictor such as the PHT with a complex predictor designed to leverage extended branch histories to make accurate predictions. For example, TAGE-based predictors [129, 131] use several tagged predictor banks to store predictions; these banks are indexed simultaneously using the branch address combined with increasing branch history sequences. To further increase accuracy, newer TAGE variants combine additional small but effective structures such as a loop predictor [130] and statistical corrector [133, 132]. The Perceptron predictor [68] perform

directional prediction using simple form of learning where the actual outcome of a branch is used to adjust weights for predictions, akin to a simple neural network. The hash-based Perceptron predictor [141] uses hashed indexing to reduce area requirements while maintaining prediction accuracy.

2.2 Transient Execution Attacks

Several key types of transient execution attacks [31, 8] originate from vulnerabilities that allow attackers to manipulate branch predictor decisions. These attacks begin with poisoning CPU branch predictor with specific branch execution history to cause either indirect branch collisions (Spectre variant 2) or mistrained conditional branches (variant 1) in victim space [79]. The BPU misprediction further leads to speculative execution along an incorrect path of instructions, also known as gadgets. While the malicious instructions executing in transient mode cannot change the architectural state (e.g., write into memory), they can still leave detectable patterns inside microarchitectural components such as CPU caches. These patterns are not cleared with the rolled-back when the misprediction is detected. A sophisticated attack can be constructed where BPU is poisoned in such a way, triggering CPU first reads sensitive data, then reveals it by leaving detectable traces in microarchiectural structures.

Not all branch mispredictions allow for transient execution attacks. A branch must be unresolved for a number of cycles to allow transient instructions from the wrong execution path to access sensitive data and leave traceable instances by initializing cache accesses. The number of instructions executed in this way, before the branch is resolved, is known as the width of speculative window [54]. Wide speculative windows are created if the information required for the branch resolution is stored in RAM. In this case, a branch can stay unresolved for hundred of cycles [96]. There are two distinct scenarios that create dangerous speculative windows. (1) When the data that determines conditional branch direction (taken or non-taken) is not located in CPU caches, and the BPU mispredicts its direction. (2) When the target of an indirect branch is not in CPU cache while BTB

contains an incorrect target due to a collision with another branch. These two scenarios describe Spectre variants 1 and 2 accordingly [79]. The second type (variant 2) of transient execution is potentially more dangerous since it allows the attacker to choose what code will be speculatively executed by poisoning the BTB. Moreover, in such an attack, the attacker can force transient execution to operate in the return-oriented-programming [134] fashion, allowing execution of instructions not present in the original binary [79]. In this work, we study this type of transient execution attacks.

2.3 BPU Centered Attacks

	Reuse-based (RB)		Eviction-based (EB)	
	Home effect (HE)	Away effect(AE)	Home effect (HE)	Away effect (AE)
Attack steps	BTB: 1. **V**: `jmp` $s \rightarrow d$; BTB $\leftarrow (s, d)$ 2. **A**: `jmp` $s \rightarrow d'$; (s, d) reused 3. **A** sees misprediction PHT: 1. **V**: `jt` $s \rightarrow d$; PHT $\leftarrow (s, t)$ 2. **A**: `jnt` $s \rightarrow s + 1$; (s, t) reused 3. **A** sees misprediction RSB: 1. **V**: `call` $s \rightarrow d$; RSB $\leftarrow (s + 1)$ 2. **A**: `ret` $\rightarrow s'$; $(s + 1)$ reused 3. **A** sees misprediction	BTB: 1. **A**: `jmp` s→d 2. **V**: `jmp` s→d' 3. **V** speculatively executes d PHT: 1. **A**: `jnt` $s \rightarrow d$; PHT $\leftarrow (s, t)$ 2. **V**: `jt` $s \rightarrow d$; (s, nt) reused 3. **V** speculatively executes $s + 1$ RSB: 1. **A**: `call` $s \rightarrow d$; RSB $\leftarrow (s + 1)$ 2. **V**: `ret` $\rightarrow s'$; $(s + 1)$ reused 3. **V** speculatively executes $s + 1$	BTB: 1. **A**: `jmp` s→d; BTB $\leftarrow (s, d)$ 2. **V**: `jmp` s'→d'; BTB$\leftarrow (s', d')$ $\mid H(s) = H(s')$, (s, d) is evicted 3. **A** sees s mispredicted PHT: *PHT entries are not evicted* RSB: 1. **A**: `call` $s \rightarrow d$; RSB $\leftarrow (s + 1)$ then fills RSB 2. **V**: `call` $s' \rightarrow d'$; RSB $\leftarrow (s' + 1)$ evicting $(s + 1)$ 3. **A** sees misprediction	BTB: 1. **V**: `jmp` $s \rightarrow d$; BTB $\leftarrow (s, d)$ 2. **A**: `jmp` $s' \rightarrow d'$; BTB$\leftarrow (s', d')$ $\mid H(s) = H(s')$ 3. **V**: CPU uses static prediction PHT: *PHT entries are not evicted* RSB: 1. **V**: `call` $s \rightarrow d$; RSB $\leftarrow (s + 1)$ 2. **A**: overflows RSB by looping `call` $s' \rightarrow d'$ 3. **V**: CPU uses static prediction
Adversarial effects	Soure and target branch addresses and calls, taken/nontaken patterns[11, 44, 86, 42, 83]	Timing channel due to **A** controlling predictions in **V** [11], speculative execution attacks [79, 83, 33, 93, 128, 164]	**V**'s `jmp` taken/nontaken[11] and `call` pattens, branch instruction virtual address[79]	Timing channel due to **A** forcing static default predictions[11], speculatively execute gadget at static prediction address[30]

A: attacker; **V**: victim; `jmp` $s \rightarrow d$: jump from s to d; `call` $s \rightarrow d$: call function d from callsite s; BTB/PHT/RSB$\leftarrow (s, d)$: store target d for branch s in BTB/PHT/RSB; $H()$: BTB/PHT hash function; $s + 1$: next instruction after s

Table 2.1: Attack surface classification for BPU collision-based attacks by event and adversarial effect types

BPU can be manipulated to enable attacks of different types. For example, an attacker can attempt to passively observe and recover branch instruction patterns. This happens during side and covert channel attacks. On the other hand, an attacker can actively manipulate the BPU state by executing branch instructions. Such a state triggers a malicious speculative execution causing data leakage. Moreover, attacks range based on what BPU property they utilize. First, there are attacks that exploit the most fundamental principle of BPU to make predictions based on the previous behavior of a branch. E.g., if a conditional branch was taken 100 times in a row, it is likely to be taken the next time. An example of such an attack is Spectre-v1 [79]. Second, there are attacks that exploit

branch collisions (aliasing). Collisions appear when two different branch instructions map into the same BPU entry and affect one another's behavior. In this work, we focus only on collision-based attacks. We believe that mitigating them is an important task on its own for a number of responses. i) There exist a large number of well-documented collision attacks that have truly devastating effects on security [164, 33, 30, 20, 104]. ii) Protecting from non-collision attacks requires different principles, such as delaying speculative execution [124] or limiting its observability [162]. iii) Even in systems that implement safe speculation, branch collisions can still happen, causing side channel attacks. Because of that, we believe protecting from collision and non-collision attacks are two orthogonal tasks.

There are two BPU features that are present in nearly all CPUs that make collision-based attacks possible. First, the BPU data structures are shared among all software executed on a CPU core, enabling branch collisions between different processes. Second, the BPU operates with compressed virtual addresses. For instance, out of 48 bits of branch virtual address, only 30 are utilized. Then, these bits are further compressed [79]. This allows branch collisions to appear within the same virtual address space, e.g., collisions between different branches in kernel and user process [42]. The deterministic nature of the BPU makes it possible for an attacker to trigger collisions in a controlled way. Our proposed solution aims at eliminating such determinism to prevent attacks.

We present a detailed overview of the collision-based attack surface in Table 2.1. First, we classify attacks by where adversarial effect takes place, either within the attacker's code (home effect) or in the victim's code (away effect). Secondly, we classify by the kind of the effect. A collision in BPU structures results in either data placed by another software entity being reused, or such data will be evicted and replaced. We refer to these as reuse-based and eviction-based attacks correspondingly. The table summarizes attacks caused by BPU collisions and their steps. Please note that there can be different adversarial effects enabled by same type of collision. For instance, a collision in BTB between two different branches can result in i) BTB-data reuse, ii) BTB-eviction and iii) activating malicious speculative execution. While i) and ii) results in side channel leakage of branch-related information

iii) is used as part of speculative execution attack to reveal victim's memory contents. As can be seen from the table, there is a diverse range of dangerous collision-based attacks. A solution like STBPU, while eliminating collision-based BPU attacks, can substantially improve security properties of microprocessors.

2.4 Existing Protections of BPU Transient Execution Attacks

Recently, several countermeasures have been developed to mitigate transient execution attacks. The majority of the proposed techniques focus on mitigating Spectre V2, as it is potentially the most dangerous variation. Although many promising protections techniques have been recently introduced by academia [70, 159, 151, 54, 51, 75], current systems are mostly protected by a few techniques developed by hardware manufacturers and software vendors. Below we summarize a set of protections that are universally enabled on today's systems regardless of OS type. Please note that for simplicity, we focus only on Intel-based machines. Next, we explore existing secure BPU designs and their limitations in comparison with STBPU. Last but not list, we compare STBPU protection and the countermeasures using other defense vectors.

2.4.1 Retpoline Sequences

Spectre v2 attacks require an indirect jump or call instruction to create a wide transient execution window. A simple compile-time solution proposed by Google [146] is to replace all indirect branches with special instruction sequences known as retpolines. These sequences emulate indirect branch functionality by pushing branch targets on stack and then executing a `ret` instruction. When predicting target for returns CPU relies on RSB instead of BTB for which poisoning is significantly more difficult [65]. Although using retpolines is considered an effective countermeasure, recent attacks on the RSB call into question the security of retpoline sequences [82, 94]. In addition, as stated by Intel, Skylake and newer processors are allowed to rely on the BTB for predicting return targets when RSB underflowing occurs [65]. This can make even retpoline-compiled binaries vulnerable.

We performed analysis to find out how common retpolines are on a typical machine. Our analysis included all executables, libraries, and kernel modules on our test machine running the most recent and fully updated version of Ubuntu. We found no retpoline compiled common executables/libraries. The kernel and a small portion of kernel modules were found to be compiled with retpolines resulting in only ≈0.06% of total binaries in the entire system being protected. This is potentially due to developers viewing retpolines as an overkill protection that results in code bloating and performance degradation [112, 137] since the system is already protected with the microcode-based protections.

2.4.2 System-wide Microcode-Based Protections

Intel responded to transient execution attacks with microcode updates introducing three new features: *indirect branch restricted speculation (IBRS)* which limits speculative execution in privileged modes, *indirect branch prediction barrier (IBPB)*, which prevents cross-process BTB poisoning, and *single thread indirect branch predictors (STIBP)*, which prevent BTB poisoning across hyper-threads [66]. We reveal more detail of microcode-based mitigations in section 4.7.1.

It is important to note that microcode-based protections do not completely eliminate the threat from transient execution. They are designed to protect from known attack scenarios while minimizing performance overhead. For instance, while IBRS by principal is capable to completely disallow speculation of indirect branch targets and thus dangerous transient execution, due to very high performance overhead it is only enabled for kernel, kernel modules, and SGX enclaves on most systems [122]. Similarly, IBPB, together with STIBP, can disallow BTB poisoning between processes and threads, but currently is enforced selectively after performing context switch into a sensitive process [66].

2.5 Transient Trojan Motivation and Current Attack Surface

We argue that the currently used protection model still leaves possibilities for attacks. Figure 2.2 demonstrates a typical attack surface of a fully patched system denoting attack

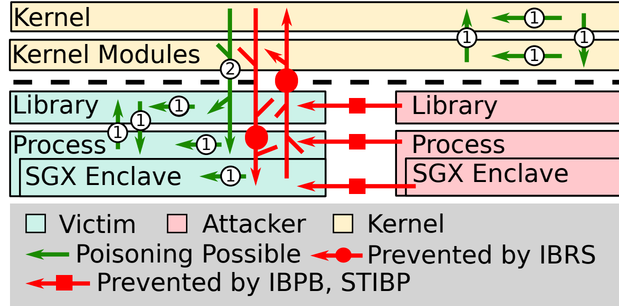


Figure 2.2: Transient execution attack surface

vectors still remaining active. Arrow tail indicates attacker branch, and arrowhead indicates victim branch locations. Two vectors are particularly useful for constructing transient trojans, denoted by ❶ and ❷ in the figure. ❶ is possible because neither IBPB or STIBP can protect against scenarios in which the poisoning branch and the branch being poisoned are located within the same address space. In Section 3.2.3.1, we demonstrate how such collisions can be easily created by leveraging newly discovered collision patterns. ❷ is possible because IBRS protects only the code running in privileged modes from being influenced by unprivileged code [1]. This permits kernel to poison the BTB and trigger malicious transient execution inside user process. We explore trojans based on this phenomenon in Section 3.2.2.

2.6 STBPU's Motivation and Protection Scope

Figure 2.3 lays two main groups of BPU-centered speculative execution attacks: collision-based e.g., spectre-w2 and history-based e.g., spectre-v1 utilizing different BPU properties.

The first group exploits branch collisions (aliasing) that appear when two branch instructions located at different addresses map into the same BPU entry and affect each other's behavior (target). As depicted in Figure 2.3 in dotted circles, existing secure BPU designs and enhancements tend to resolve these attacks but all have incomplete protection coverage or unsupported performance. We provide detail in §2.4.

[1]IBRS implementation may vary between CPU generations and OS policies enabling or disabling this vector

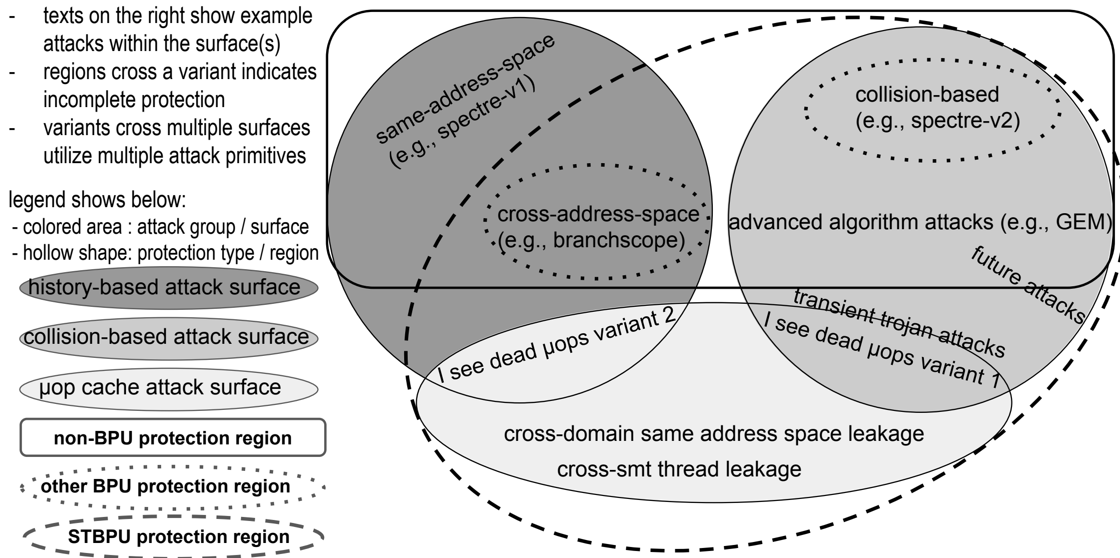


Figure 2.3: STBPU protection region v.s. the others

The second group exploits the most fundamental property of BPU, to record branch's statistics and then activate speculative execution based on such statistics. This makes BPU trainable by attackers via providing branch execution history. These attacks are often mitigated by non-BPU protections, including secured cache designs [115, 156, 153]; shadow structures to hide the effects of erroneous speculative executions [71, 160]; tracking speculative data flow for invisible speculations [162, 116, 88] and offline gadget detectors [105, 55], etc.

Depicted as solid square in Figure 2.3, non-BPU protections trying to cover both groups of attacks often introduce high costs on performance and resource. More importantly, they overlook BPU as an attack vector i.e., BPU is still prone to collisions, resulting critical penalty in security. For instance, researchers recently discovered micro-op cache (also called decoded stream buffer, DSB) as disclosure primitive [123]. Depicted as lightest grep area in Figure 2.3, This new timing channel can leverage BPU for more powerful transient execution attacks that does not require implicit or explicit data access, bypassing several existing mitigations such as STT [162] and context-based fencing [140] and voiding their security guarantee on both spectre-v1 and spectre-v2.

The incompleteness in protection motivates us to propose, STBPU, a safe BPU design that is immune to collision-based attacks and BPU side channels and lifts non-BPU protections' inherent overhead towards all-around security. As depicted as the dash-dotted circle, STBPU focuses on protecting collision-based attacks but is not limited to it. The aforementioned new μop-cache attacks are interesting cases, showing the STBPU design can mitigate more powerful spectre-v1 variant.

In i see dead μop attack variant 2 [123], an attacker first prime certain micro-op cache sets and performs Spectre-v1 type of mis-training on a authorization check and a followed transmitting indirect branch instruction in a victim method. Next, attacker enters untrusted input. The authorization check will fail. If enabled, non-BPU mitigations such as fencing or speculative data flow control will restrict the followed instructions from dispatch or being visible. However, they do not prevent this secret-dependent indirect branch from being speculatively fetched, further leaving a footprint in the micro-op cache. As a result, the attacker can probe the micro-op cache set and infer the secret. On the other hand, such attacks become inapplicable with the STBPU design of DSB. This is due to ST remapping makes DSB indexing non-deterministic. As a result, on both priming and probing stage, the attacker loses the control on both the entry branch address and DSB set mapping even within the same address space.

DSB caches stream decoded micro-ops from multiple decoders after the branch prediction stage, making it naturally applicable to use ST remapped address for its own indexing without additional latency. As DSB is enabled only through branch [67], a consistent isolation can be maintained via ST remapping and ST re-randomization. We analyze similar mechanism with more complicated BPU attacks in §4.5. Since STBPU substructure modification will be detailed in §4.3, we omit the DSB discussion detail as it is similar and simpler.

History-based attack such as Spectre-v1 leverages the correlation between the natural consequence of prediction and the unsafe follow-up front-end acceleration. Thus, we believe Spectre-v1 mitigation should be outside of BPU such as by existing non-BPU protections,

an orthogonal safe speculative execution control, or a safe loop stream detection (LSD) unit. STBPU benefits these protections by largely reduce their surface of enforcement, granting better performance.

STBPU will focuses on defend against collision-based attacks and BPU side-channels. Evidenced by a large number of well-documented dangerous exploits, they are responsible from critical data leakage to speculatively executed arbitrary code. Categorized by their mechanisms in §2.3, we will analyze the security STBPU against these dangerous exploits in §4.5.

Chapter 3

Branch Predictor & Transient Execution Trojans

3.1 Introduction

Increased performance of modern processors largely relies on various hardware units performing activities ahead of time. For example, when the processor encounters a branch instruction, a type of instruction that alters the normal sequential execution flow, the branch prediction unit (BPU) predicts the address of the following instruction instead of waiting for the correct address to be computed. In order to avoid damaging the architectural state, execution based on predicted data is performed in a special *transient* (or speculative) mode, which permits roll-backs to previous states. If the prediction is correct, the execution along the predicted path continues. Otherwise, the CPU reverts any changes made by executing incorrect instructions. Recent transient (or speculative) execution attacks, including Meltdown [89] and Spectre [79], demonstrated how such performance optimizations can be manipulated to force victim programs to leak sensitive data by leaving detectable traces in microarchitectural data structures such as CPU caches. These attacks are capable of violating the most fundamental principles of memory safety, including user-kernel isolation. From early 2018, these attacks opened up a new class of microarchitectural threats and

quickly spawned many variations [31, 150, 77, 94, 82, 128, 44].

Numerous mitigation techniques have been proposed to protect from transient execution attacks. These techniques range from serializing instructions [78, 66, 4], avoiding dangerous code sequences [5], flushing hardware data structures [2, 4], and limiting transient execution [159, 70] to disabling microarchitectural covert channels [35, 29, 75, 45]. We provide a more detailed description of current protection schemes in Section 2.5. Hardware manufacturers, including Intel and AMD, responded to the threat of transient execution attacks with a series of microcode updates. While being effective in mitigating the main problem, such microcode-based countermeasures noticeably reduce performance [112].

In this work, we argue against the widely spread perception that the triggers and effects of transient execution attacks are fully understood, and recommended protections leave no room for any attack to occur. We do so by constructing transient trojans. These malicious software modules conceal their malign functionality in transient execution mode, and unlike previously demonstrated attacks [79, 150, 77, 33], do not require an external attacker controlled process to activate the hidden functionality. First, we perform a reverse engineering study of branch predictor mechanisms in recent Intel and AMD processors and discover several new branch collision triggering techniques. These techniques enable portable, self-contained trojans that can be included in sensitive software (for instance, by a malicious open-source project contributor). Then, we construct software modules that encapsulate all attack components (poisoning and victim branches) inside a single process. Malicious functionality concealed in transient execution mode can remain unnoticed in software even after undergoing rigorous security checks such as symbolic execution [19], taint analysis [36], model checking [47], various methods to detect traditional software backdoors [136, 142, 127, 125, 157], and even existing Spectre detection tools [5, 54, 151, 3]. According to recently proposed transient attack classification by Canella et al. [31], transient trojans described in this work present a practical example of the same address space transient execution attacks. We argue that transient execution ubiquitously present in nearly all today's CPUs is a natural fit for concealing malicious code since it offers an execution mode

that is completely invisible to existing binary and source code analysis techniques.

In summary, this work makes the following contributions:

1. We perform a reverse-engineering study[1] of the BPU to uncover the mechanisms responsible for indirect branch prediction and ways to manipulate them. This allows us to construct three types of trojans, each relying on a different BPU anomaly.
2. We present a new branch instruction collision mechanism based on early BPU accesses. First, the mechanism allows attackers to construct trojans that can avoid being detected by current techniques based on code analysis. Second, it permits creations of small and portable trojans.
3. We propose a technique to disperse transient gadgets, improving their stealthiness and effectiveness.
4. We analyze the static prediction mechanism and conclude that it can result in skipped indirect branches, which we use to bypass existing gadget detection techniques and to construct trojans.
5. We present an analysis of current binaries that demonstrates a high prevalence of potentially dangerous collisions reaching hundreds of thousands in large binaries. We argue that such naturally occurring collisions can be used to hide malicious trojans as well as constructing trojans from existing code.
6. Finally, we analyze protection techniques and suggest approaches to remove the threat from uncontrolled transient execution.

Responsible Disclosure Research findings in this work have been reported to Intel and AMD.

3.1.1 Threat Model and Assumptions

We assume that the attacker is a malicious developer who is capable of delivering software that seems benign before being activated by a trigger condition. The user (victim) may run

[1]Experiments were performed on Intel Haswell (i7-4800MQ), Skylake (i7-6700K), Kaby Lake (i7-8550U), and AMD Ryzen (1950X) machines running recent and fully patched Ubuntu OS with microcode patches installed.

static or dynamic analysis and information flow control tools. Moreover, for trojans based on newly discovered collision patterns (Sections 3.2.1 and 3.2), the user can run existing Spectre gadget detection tools [5, 54, 151, 3]. The malicious code can be distributed in the form of a precompiled binary, source code, a shared library, or a commit to an open-source project. We assume the attacker has general knowledge about the configuration of the victim's machine, such as CPU microarchitecture generation, versions of shared libraries, and kernel.

3.2 Transient Execution Trojans

In this section, we present transient trojans, programs that can compromise security while containing no malicious instruction sequences in any place reachable by normal execution flow. Even though these trojans appear benign, they output sensitive data when malicious transient execution is activated. The basic building block for a trojan is a condition in which transient execution temporarily violates the architectural state of a program. One of such violations is when two branch instruction collide in BTB. As a result, the body of one branch is executed with data in registers from another branch. This enables a basic memory safety violation, which can lead to sensitive data leakage.

In this section, we describe reverse engineering of mechanisms used to predict indirect branches. We introduce three distinct types of trojans, each utilizing a different kind of BPU anomaly. We show that a malicious developer or an open-source contributor can compose a self-contained software module in which malicious functionality is concealed in transient execution. Unlike previous works [79, 150, 77, 33], which require a separate malicious process controlled by the attacker for BTB poisoning, our self-contained trojans could combine all attack components, including BTB poisoning, in one single process.

3.2.1 Branch Target Prediction Mechanisms

Modern BPUs are capable of predicting both direct and indirect branches with high accuracy. The mechanisms for predicting targets of these two branch types differ substantially.

Figure 3.1 demonstrates a simplified target prediction mechanism overview. Since the target of a direct branch (including direct calls, jumps, and conditional branches) is fixed, it is predicted by BPU simply caching previously calculated target and storing it in a set-associative BTB [111]. As in any set-associative cache, each lookup is done using *index*, *tag*, and *offset* bits. Index bits determine BTB set for the lookup, while tag and offset allow selection from multiple entries in the same set. To predict the target of a direct branch, the BPU performs a simple lookup based on the branch source address. The address bits are typically hashed to reduce the number of bits stored as tag in BTB.

However, this mechanism is not sufficient for effectively predicting indirect branches because a single indirect branch may jump to different destinations depending upon data the program is processing. Thus a prediction mechanism must account for the context in which the branch is executed. Current BPUs do so by associating indirect branches with patterns of previously executed branches. This is achieved using the mechanism called the branch history buffer (BHB), a shift register structure that serves the purpose of accumulating the branch context. The context is composed by hashing addresses bits of every committed branch instructions with current BHB value [58]. Then compressed BHB value is used to perform target lookups. Such a predictor allows storing multiple targets for a single indirect branch and accurately predicting targets in cases when they depend on previous code sequences.

To maximize the utilization of the BPU storage resources, instead of storing targets for direct and indirect branches in separate structures, both predictors share a single large BTB as in hybrid predictors [32]. The two predictors differ by the type of BTB addressing modes they use: instruction-pointer based (IP-based) and branch history buffer based (BHB-based).

In IP-based addressing, the index, tag, and offset for a BTB lookup are calculated solely based on a subset of the branch instruction virtual address bits. This mode is primarily used for direct branches.

In BHB-based addressing, the lookup is performed based not only on branch instruction

address but also on the state of BHB. For instance, compressed BHB value can be used as the BTB tag, allowing to store multiple targets for a single indirect branch. This mode is exclusively utilized by indirect branches. However, when BPU is processing an indirect branch, the two predictors are used concurrently with the prediction selected based on accuracy monitoring for each entry stored in BTB. We provide details further in this section.

While finding an entry based on index calculating and tag matching reminds a normal cache operation, BTB operates differently compared to regular caches. We performed a reverse engineering study to understand the BTB configuration and how branch address bits are used for lookups. We use direct branches to study the IP-based addressing mode. In the first step, we observe that, on Skylake processors, only 30 least significant bits from the branch source address are used for lookups, and the bits [47:30] are ignored, confirming results of previous studies [42]. Then we determine the associativity of the BTB. Assuming bits from the most significant chunk of the remaining [29:0] are used as tag, we create n branch instructions with mismatching tags by flipping these bits. We keep other address bits identical to make matching index and offset. We make each of these branches having a non-matching target. Then we execute this branch sequence twice, observing BTB miss events for any of them during the second time. We use hardware performance counters [1] to detect BPU events. A BTB miss indicates the BTB does not have enough ways in a given set to store all n targets resulting in eviction of one of the targets. We observed no misses for $n < 9$ and a stable miss pattern when $n \geq 9$, indicating that BTB contains 8 ways.

Next, we find which address bits are used as index. To do so, we execute a set of 8 branches that occupies an entire BTB set. Then, one more branch is executed while flipping its address bits in range [29:0]. If the flipped bit is used as tag, all 8+1 branches will have identical indexes and be assigned to the same set. In such case, one of the 8 targets will be evicted. However, if the flipped bit is used as index, the branch with the flipped bit will go into a different set, and no evictions will appear. Then we check if any of the 8 branches were evicted from BTB. This way, we identify that bits [13:5] are used as index providing 2^9 total

sets resulting in 4 096 total BTB entries. This suggests bits [29:14] used as tag. Previous research [58, 79] determined that tag bits are folded together using a simple XOR operation: `tag`$= a_i \oplus a_{i-8} | i \in [29, 22]$, where a is the branch instruction address. We observed that the exact addressing scheme and bit folding function varies on different microarchitectures. For instance, Haswell processors appear to use `tag`$= a_i \oplus a_{i-9} | i \in [30, 22]$ folding function.

Finally, the remaining bits [4:0] are used as the offset. The exact role of the offset in the context of BTB is unclear. However, the presence of offset is indicated by multiple sources [59, 73, 15]. In general case, the offset can be viewed as a second tag requiring a full match to produce a BTB hit. However, as we discover in Section 3.2.3.1, the matching is done using a more complex function, which can produce additional collisions resulting in potentially malicious transient execution.

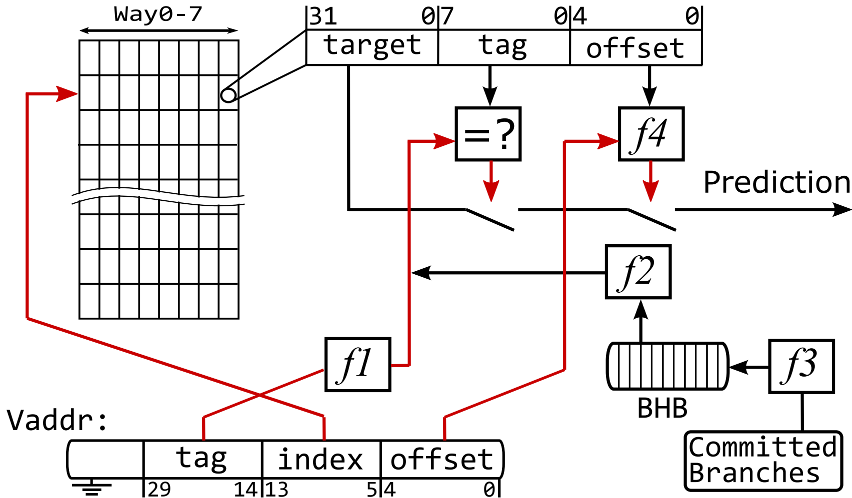


Figure 3.1: Branch target prediction mechanism combining direct and indirect branch prediction logic. Functions $f1-3$ are bit compression functions; $f4$ is bit matching function. Mechanisms used for trojan construction are highlighted red

3.2.1.1 Addressing Modes for Indirect Branch Prediction

Predicting indirect branches based on the context in which they are executed is a logical strategy. Consider a `switch-case` expression in C. It is typically implemented by calculating the resulting target and jumping to this target via an indirect jump instruction. The code pattern executed prior to the `switch` is likely to affect which target will be taken. For this

scenario, the BHB-based prediction mechanism is accurate. However, many `switch-case` expressions also have the `default` case, a single target for multiple different (unrecognized) contexts. In this case, the BHB-based mechanism will not be optimal. Instead, the simple IP-based approach will correctly predict the same target regardless of the context. We hypothesize that BPU uses both mechanisms concurrently.

To verify our reasoning, we designed an experiment in which *the same* indirect branch is executed in multiple different contexts. The contexts are created by varying taken-not-taken patterns of preceding 50 conditional branches. Our experiment included the following contexts: $A \rightarrow a$, $B \rightarrow b$ and $R_{1..k} \rightarrow r$, where $\{A, B, R_{1..k}\}$ are branch contexts and $\{a, b, r\}$ are target addresses for each corresponding context. Contexts A and B have their own targets, while k contexts share a common target r. Executing an indirect branch in different contexts while observing its misprediction rate via hardware performance counters allows us to detect when each addressing scheme is used. For instance, a pattern `ABABAB` has mispredicted branches for the first two times and correct predictions (hits) for the following ones. This is because the branch predictor quickly learns the dependency between context A and target a and between B and b.

Table 3.1 presents experimental data collected from running two demonstrative patterns 1 000 times and averaging the results. The first pattern shows how after the branch is executed for the first time, the predictor learns its target to be a. Because of that, it mispredicts the target when we execute it in context $R_1 \rightarrow r$. However, any consequent execution in random context $R_n \rightarrow r$ is correctly predicted to go to r. It also shows how the branch is correctly predicted when we execute it in static context $A \rightarrow a$ second time. These observations show that the branch predictor is capable of predicting the same branch instruction using two independent modes.

The second pattern demonstrates how two addressing modes work in parallel; i.e., the predictor simultaneously checks whether a branch is available using either of the schemes. If it finds a matching tag using any of the schemes, it proceeds with the stored target. In Figure 3.1, we demonstrated BPU design that can produce such behavior.

These observations allow us to identify two distinct types of indirect branch collisions. Type 1 collisions are when both the BHB state and the reduced branch source address are matched, and the BPU uses BHB-based addressing. Type 2 collisions are when only the branch addresses are matched while mismatching the state of BHB, and the BPU uses IP-based addressing.

	Pattern 1						Pattern 2							
Pattern	A	R_1	R_2	R_3	A		A	R_1	B	R_2	A	R_3	B	R_4
Observation	M	M	H	H	H		M	M	M	H	H	H	H	H
Miss rate	0.99	0.99	0.0	0.0	0.05		0.99	0.99	0.99	0.02	0.14	0.0	0.04	0.0

Table 3.1: Misprediction rate observed in two different patterns composed by varying the BHB context. H represents hit, and M represents misprediction

3.2.1.2 Selecting Branch Type for BTB Poisoning

Previous attacks based on BTB poisoning [79, 150, 77, 33] used type 1 collisions. In these works, a victim branch was poisoned from a different process by executing an indirect branch on matching virtual addresses while mirroring the BHB state via repeating behavior of preceding branches. Such setup is less suitable for constructing real-world transient trojans since they must be self-contained; the branch performing poisoning and the branch being poisoned must be located within the same address space. From now on, we refer to the former as **writer branch** or **WB**, and the latter as **reader branch** or **RB**. To construct a trojan based on type 1 collisions, an RB and a WB must be placed at the addresses producing collisions, and have identical BHB states when executing. This is a challenging task due to mapping function $f2$ and BHB update function $f3$ (from Figure 3.1) unknown or partially reverse-engineered [58, 79]. Even if these functions are fully reverse-engineered, BHB training would require highly irregular code sequences that can be easily detected.

Intuitively, using type 2 collisions is a better option. However, collisions of this type require that both an RB and WB are executed in a new BHB branch context each time. This can be done by running sequences of random taken/not-taken conditional branches before executing WB and RB. This is problematic because unique BHB states will eventually start to repeat, forcing the BPU to switch to the BHB-based mode of addressing. In addition,

such code would be highly irregular.

A desired mechanism for constructing trojans must 1) produce reliable collisions when RBs and WBs are located in the same address space; and 2) be easy to mask as benign code. We propose to use direct branches as WBs since 1) they are always handled by the simple IP-based addressing mode making BTB writes more deterministic; and 2) they are common in regular applications with approximately every 4-7th instruction being a direct branch making them easy to mask as normal code.

3.2.1.3 Finding Branch Collisions

We hypothesize that the mechanism used to predict direct branches is exactly the same as the IP-based addressing mode for indirect branches. If this hypothesis is true, constructing a trojan becomes straightforward. If we match the address bits used for the tag, index, and offset in a direct WB and an indirect RB, the WB will poison the RB. This will result in speculatively executing code pointed by WB's target when the CPU processes the RB.

To verify this hypothesis, we design an experiment depicted in Figure 3.2, which allows to reliably identify addresses that result in branch collisions. In this experiment, a direct jump instruction located at address `addrWB` jumping to `addrT1` acts as a WB. An indirect jump is located at `addrRB` jumping to `addrT2` acts as an RB. Then we place a transient gadget at address `addrT3`. This gadget accesses a variable `dat`, loading it into CPU's data cache. If the two branches collide, then mispredicted RB results in transient execution going to `addrT3`, activating the gadget which loads the variable `dat` into the cache. We detect RB mispredictions using hardware performance counters while measuring the latency to access `dat` tells us if the gadget was activated. By moving these branches and gadget instructions in virtual address space and observing collisions, we can effectively scan address space to find addresses that create collisions and analyze corresponding target calculation mechanisms. Using this setup, we make several important observations.

Observation 1: Direct branches can serve as WBs, and indirect branches can serve as RBs creating ideal grounds for trojan construction. Moreover, indirect RBs do not need to

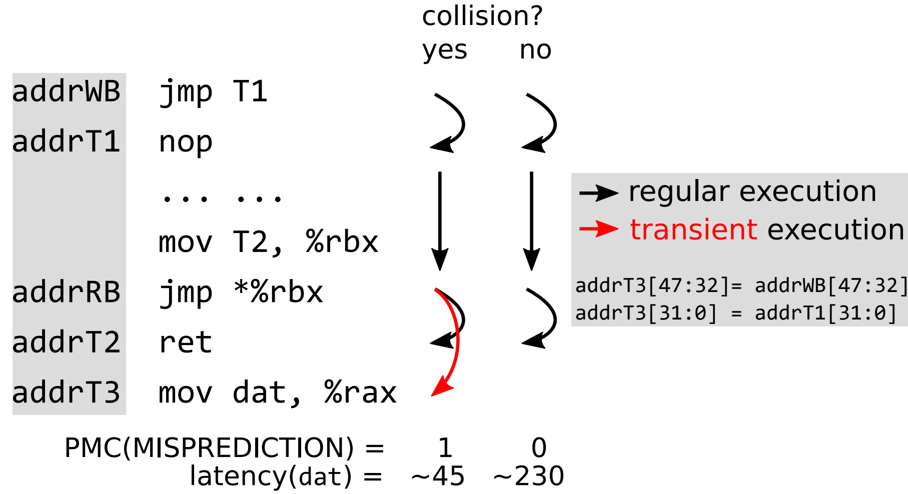


Figure 3.2: Collision detection experiment setup

be executed in a new context every time, as explained in Section 3.2.1.1.

Observation 2: Reduced data stored in BTB (tag and target bits) allows to create collisions within a single process and redirect execution to malicious address. For instance, BTB stores only 32-bit target [79], and to compose the 48-bit prediction target, the CPU simply concatenates branch source address bits [47:32] with the 32-bit target from BTB. This enables attackers to use relative addressing.

Observation 3: We tested different types of branch instructions and concluded that any direct branch can serve as a WB, including calls and conditional jumps.

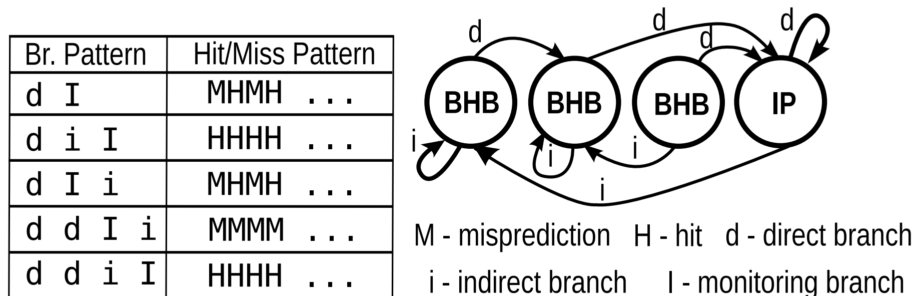
Observation 4: Our initial tests demonstrated a 50% rate of successful poisoning. However, this rate can be improved if direct a WB is executed multiple times, indicating the possibility of a tournament mechanism [53] selecting the most accurate predictor.

3.2.1.4 Predictor selector mechanism

To investigate the nature of observation 4, we conduct the following experiment. We place an indirect branch (**i**) and a direct branch (**d**) at colliding addresses and make them having mismatching targets. Since **d** always uses the simple IP-based addressing mode, the BTB will contain an incorrect target when predicting **i** using this mode. By preceding **i** with a fixed sequence of conditional branches, we guarantee identical BHB states. As a result,

BHB-based mode will always produce a correct prediction. If a tournament mechanism is present, we expect the predictor selector mechanism being affected by executing both of the branches. In particular, when we execute **d**, it will be correctly predicted using the IP-based mode. This will increase the confidence of this mode. In contrast, when **i** is executed, a misprediction from the IP-based mode and a hit from the BHB-based mode will decrease the former and increase the latter predictor's confidence.

Observing **i**'s correct/incorrect prediction patterns allows to detect which predictor is currently in use. A mispredicted observation indicates the IP-based mode usage, while correctly predicted branch tells BHB-based mode is in use. By executing sequences composed from these two branches and observing **i**'s prediction accuracies, we can detect when each predictor wins the tournament. We execute patterns created by invoking **i** and **d** in a random order while collecting the misprediction patterns. Figure 3.3 demonstrates our observations from several characteristic repeated patterns. Please note that demonstrated prediction patterns are from a single execution of **i** (denoted by capital I) in multiple rounds. By manually inspecting these patterns, we noticed that the observed behavior resembles a finite state machine (FSM) implemented using a 2-bit counter as the system appears switching between 4 stable states. It is possible to manipulate such a mechanism. For instance, executing **i** multiple times in a row increases the accuracy of the BHB-based predictor and makes it more likely to be used for future branches.



Br. Pattern	Hit/Miss Pattern
d I	MHMH ...
d i I	HHHH ...
d I i	MHMH ...
d d I i	MMMM ...
d d i I	HHHH ...

Figure 3.3: Misprediction patterns demonstrating the competition between the two addressing modes and an FSM matching this behavior

In the effort to find the configuration of the FSM responsible for such behavior, we

performed the following analysis. First, utilizing the brute-force approach, we generated all possible FSM configurations based on a 2-bit counter. The states of the FSM determine which predictor addressing mode (IP or BHB based) is used. This resulted in 863 040 possible configurations. After removing configurations containing infinite loops and other abnormalities, we reduced this number to 49 104. Next, we simulated these FSMs and ran previously collected patterns through them while observing which predictor is utilized each time. During this stage, we only keep the FSM configurations that match the real system behavior, resulted in only 6 possible unique FSM configurations. We present one such potential FSM in Figure 3.3. Please note, while this FSM configuration is capable of modeling the real system behavior with high accuracy, the actual mechanism used in the CPU may be different. Knowing the inner workings of the predictor selector, an attacker can perform manipulations forcing the CPU to use the IP-based prediction mode to enable simple collisions by triggering repeated execution of the colliding direct branch instruction.

3.2.2 Distant Collision Trojans

Now we introduce the first and most basic type of a transient trojan and demonstrate its inner workings. This type of trojan is based on exploiting the BTB addressing scheme where only partial address information is stored. This allows two distinct branches (WB and RB) to collide in a way that when RB is mispredicted, the transient execution goes to the target of WB violating the architectural state. For example, as we described earlier, the tag stored in BTB is folded using a simple XOR operation. Suppose there is a direct branch at address `0x400077` and an indirect branch at `0x4077` in the same process. These branches will collide in BTB when the IP-based addressing mode is used. The attacker can prepare a binary containing branches at colliding addresses. When the binary is deployed on the victim machine, the collision is activated by calling normal API functions in a specific order. In short, this type of transient trojans operates in the following way. First, using a program API, the WB will be activated to write the poisoning entry into the BTB. After that, the attacker trigger conditions for the RB to initialize transient execution, e.g.,

issuing an API call to access a large array forcing the RB's target to be removed from CPU cache. Then, the RB is executed, and BPU uses the poisoned BTB entry to begin transient execution of a gadget that accesses secret data and reveals secret values using microarchitectural covert channels [99, 61, 34, 49, 41, 50, 13, 72]. We assume the attacker being able to use return-oriented analysis techniques [57, 18, 109] to find or create code sequences (gadgets) that, when executed in transient mode, result in a desired malicious activity. Generally, gadgets can leak data by either 1) leaving traces in shared resources such as CPU data caches [79] or 2) by affecting the timing of certain operations in a controlled way. As demonstrated by Schwarz et al., such delays can be detected over a network [128]. In addition, this type of trojans can be constructed by placing RB and WB in different memory segments within a single application context. For instance, WB can be placed (or existing branch can be utilized) in a library or kernel code segments, while RB being located in trojan's `.text` segment.

Please note, although we construct this type of the trojan utilizing a known branch collision mechanism, we believe that our approach is substantially different. In existing works, a lower privilege entity, such as an untrusted process poisons a branch in a higher privilege entity such as an OS kernel or an SGX enclave [79, 150, 77, 33]. Such attacks are currently mitigated via IBRS, which protects higher privileged entities (kernel and enclaves) from lower privileged entities. We utilize poisoning vectors that are typically not hindered. In current systems, collisions still occur in many ways as we summarized in Figure 2.2. The two types of poisoning we will use for constructing trojans are 1) when higher privileged entity poisons a lower privileged entity and 2) when poisoning happens within the same privilege level.

3.2.2.1 Trojan example utilizing a system call

Assume a malicious developer whose goal is to construct a program that handles secret data and, when triggered, leaks this data. A typical manual inspection or static/dynamic analysis would look for any reference to the sensitive data to make sure they do not reveal

it via a covert channel [39]. To show how a practical trojan can be constructed containing no such references, we provide a simple demonstration in which poisoning is triggered by executing a benign existing system call. Performing system calls is a normal activity for any application and unlikely to cause concerns. During a system call, control is temporarily transferred to the operating system. As a result, branches residing in kernel memory trigger writes into the BTB. When the system call is completed, the execution transfers back to the trojan without removing BTB entries placed during the kernel execution. If any of these BTB entries have matching `index`, hashed `tag`, and `offset` bits with an indirect branch in trojan's code, the BPU will treat it as a hit. The predicted address will be composed by concatenating the kernel branch's 32 least significant target bits with the remaining 16 bits from the trojan branch's source address. If such an address contains executable memory, transient execution will take place until CPU detects misprediction and rolls back to the previous state. This will result in violated architectural state. We utilize this phenomenon to construct a trojan that solely relies on normal code executed during a system call to redirect transient execution to a place containing a malicious gadget within trojan's code segment.

During the trojan preparation stage, the developer performs an analysis of the environment in which the future trojan will run and finds a direct branch suitable for poisoning. Typically, this branch needs to be in the final stage of a short system call routine. For our proof-of-concept prototype, we choose a branch inside the `open()` system call. Then the developer introduces a code construction that results in an indirect branch at the colliding address while sensitive data is possible to reference (for instance, the pointer to that data is in one of the registers). This indirect branch transfers regular execution to a benign code containing no leakage instructions. As a result, static analysis will not raise any flags. Modern-day compilers offer a wide range of code constructions that are compiled into code with indirect branches such as virtual functions, function pointers, and computed `goto`s. In addition, a trojan developer can use function alignment and memory mapped code region techniques to easily achieve desired instruction placement.

The next stage of the trojan preparation is finding a suitable transient execution gadget. The gadget must first access the sensitive data and second leak its value via covert channels.

A high level schematic description of the trojan activity is depicted in Figure 3.4. For each iteration of the attack, the attacker interacts with the trojan via API calls. Each call activates the malicious function inside the trojan, which in turn performs a system call causing BTB poisoning. After the function returns from the system call, it executes an indirect jump, resulting in transient execution of the gadget. After this, the attacker probes the system to obtain microarchitectural traces and recovers leaked data. To evaluate the accuracy of this type of trojan, we collect data from 1 000 rounds of trojan execution. In each round, the gadget is triggered 1 000 times. Then, we count the number of times the gadget is successfully activated. The average success rate for this experiment is 12.79%. Such a rate is within an acceptable range for most microarchitectural attacks. To compare this result to a clean environment, we composed a prototype in which a WB and an RB are both located inside user process memory segments. The average accuracy rate for this configuration is 94.52%. Such a significant improvement is likely due to the normal side effects of system call execution inside the kernel and a mode switch. For instance, system call activity is more likely to evict gadget code from the instruction cache stopping the transient execution attack. Please note that similar trojans can also be constructed by using library functions instead of kernel code. Since library code is placed inside the process address space, IBRS will not prevent the poisoning.

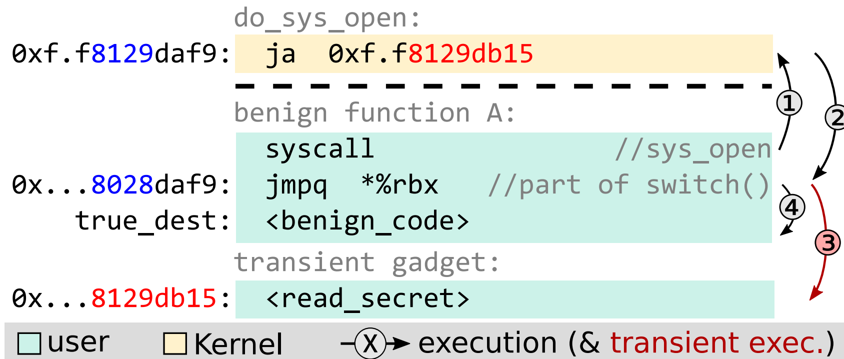


Figure 3.4: Transient trojan based on `open()` system call

Please note that ASLR and KASLR can make these attacks challenging. However, programs may be compiled without ASLR support and distributed in binary form. Even if KASLR is enabled, its entropy is very small, making attacks still possible by placing RBs at all potential collision addresses. To eliminate the dependency on hardcoded code addresses, we develop two types of portable trojans that work regardless of code placement.

3.2.3 Portable Trojans

3.2.3.1 Early Front-end Branch Collisions

Timely branch predictions are very important for the performance of CPU front-end. BPU is responsible for identifying branch instructions early and adjusting fetching to guarantee delivery of instructions from the correct execution path to minimize the number of costly roll-backs. Any slowdown in generating a prediction results in a front-end delay, which propagates into other stages of the pipeline. However, to perform a lookup, BPU needs to know the address of instruction's *last byte*. This is because, typically, BPUs address branches using their least significant byte. On a CISC processor with variable instruction length, such information is not immediately available. A special front-end component, called predecoder, is responsible for detecting instruction boundaries inside a prefetched instruction cache line. We hypothesize that modern-day aggressive front-end designs may avoid waiting for predecode to complete and activate transient execution based only on partial information about potential branch instruction address. This can result in an *early front-end branch collisions* where closely located branches collide due to uncertainty in the boundaries of branch instructions. If this is true, then collisions may appear between branches with mismatching least significant address bits. Several Intel patents [59, 73, 15] refer to these bits as offset while not explaining their exact purpose.

To test the aforementioned hypothesis, we adapted the experiment depicted in Figure 3.2 with the following changes. First, we position both WB and RB within the same 64 byte instruction cache line. This guarantees matching tag and index bits. Next, we make the direct WB jump to a gadget that now leaks a value stored in register `%rax`. Before executing

it, we always load a non-secret value in that register. The indirect RB, as previously, jumps to a benign code. However, prior to that, it loads a secret value into the register `%rax`. If the RB is poisoned by WB, the transient execution shall transfer to WB's body but with secret data loaded in the register. Finally, we execute the WB and RB in a loop and observe effects. If poisoning happens, we detect the secret value leaked via the cache covert channel. An adapted version of this experiment is demonstrated in Figure 3.7.

We use this setup to scan all possible positions of WB and RB and detect when poisoning happens. As a result, we were able to find stable collision patterns on all tested Intel processors. These patterns indicate a partial offset bits matching mechanism. In particular, on Skylake and Kaby Lake processors: *WB and RB collide either if all offset bits are matched or if bit 5 in WB address is 1 and 0 in RB address.* Thus when generating a prediction for the indirect RB, the BPU mistakingly uses the target of another branch instruction located in one of the subsequent memory locations. On Haswell, a similar pattern exists, however, with bit 4 triggering these collisions instead of 5. These patterns are demonstrated in Figure 3.5.

This intriguing pattern variation between CPU generations sheds some light on the likely root that causes this collision mechanism. To investigate it, we carefully compared microarchitectural front-end optimizations involved in early instruction processing in Haswell and Skylake processors [6]. Our reasoning is that the mechanism responsible for the behavior must be located in the pipeline before the instruction predecoder and size of instruction blocks it processes is double in Skylake compared to Haswell.

By carefully examining related front-end components [6], we concluded that the decoded streaming buffer (DSB) [121, 7] is a potential root cause. In Intel processors, DSB (also referred as μop cache) helps to avoid decode/predecode delay by storing ready to execute microcode operations (μops). The most performance benefit comes from situations where instruction decoding is delayed, for instance, due to an instruction cache miss or decoders being busy. It also reduces power consumption by suppressing overall decoder activity [138]. Branch prediction while executing μops stored in DSB is equally important for performance

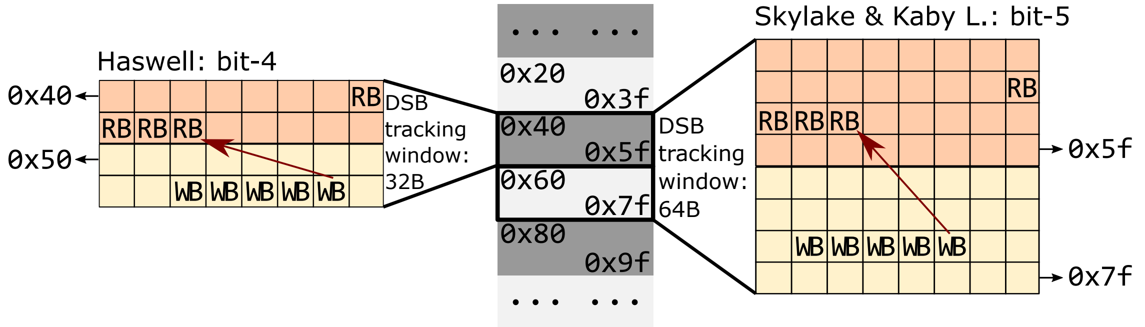


Figure 3.5: Branch collision patterns within the same cache line on Haswell and Skylake CPUs

as it can trigger μops dispatched directly from the DSB to instruction decode queue, which naturally bypasses all the pre-decoding and decoding stages [120]. However, branch prediction in this stage is challenging due to the specifics of addressing in DSB where the virtual address of only the first instruction inside a tracking window block (32 bytes on Skylake) is stored [69]. Since a single macro instruction can be decoded into a different number of μops; entries in DSB are not aligned with regular instructions in virtual memory. Therefore, the DSB does not have sufficient information on the boundaries of a branch μop. To perform a precise BTB lookup, the DSB logic would have to compute macro-op address from the virtual address of the first μop in the DSB block and the offset. That would significantly increase the mechanism's complexity. Alternatively, DSB can request predictions without specifying the instruction location within its window. We argue that our experimental data suggests the existence of such mechanisms. Our attack example demonstrates how this premature BPU lookup can result in incorrect predictions and malicious transient execution. It is worth mentioning that the size of the DSB tracking window enlarged from 32 Bytes in Haswell to 64 Bytes in Skylake and Kaby Lake. This may explain the bit-4 and bit-5 observations on these CPUs.

Please note, that this collision mechanism initially appears less stable and is sensitive to surrounding code and branch activity of the program. The average attack success rate in a series of experiments was 4.86%. This is due to this type of collision relying on tight

race conditions and contentions inside the front-end components. We tackle this problem by developing an automated collision optimization technique based on an evolutionary algorithm approach in Section 3.3.

Please note that the collision mechanism described above also works when combined with other collisions types. For example, if two branches have `tag` and `index` bits matched while mismatching higher (ignored) bits ([47:30]), and following the bit-5 collision pattern, the collision will also occur. Figure 3.6 demonstrates this principle. Presented are results from a Kaby Lake experiment in which we placed an RB at address `0x300110` and then scanned for potential addresses where collisions can occur (`0x100300100` – `0x100300140`) whilst monitoring access time to the variable that is only accessed from transient execution. Low access latency indicates a collision happening. One such collision is between addresses `0x300110` and `0x100300110`. This is due to the full `index`, `tag`, and `offset` match. As seen from the graph, there are additional bit-5 collisions occurring when the WB crosses the 32-byte boundary, and offset collisions start taking place. For simplicity, we will refer to all such collisions as bit-5 collisions regardless of microarchitecture and whether or not they are combined with other collision patterns.

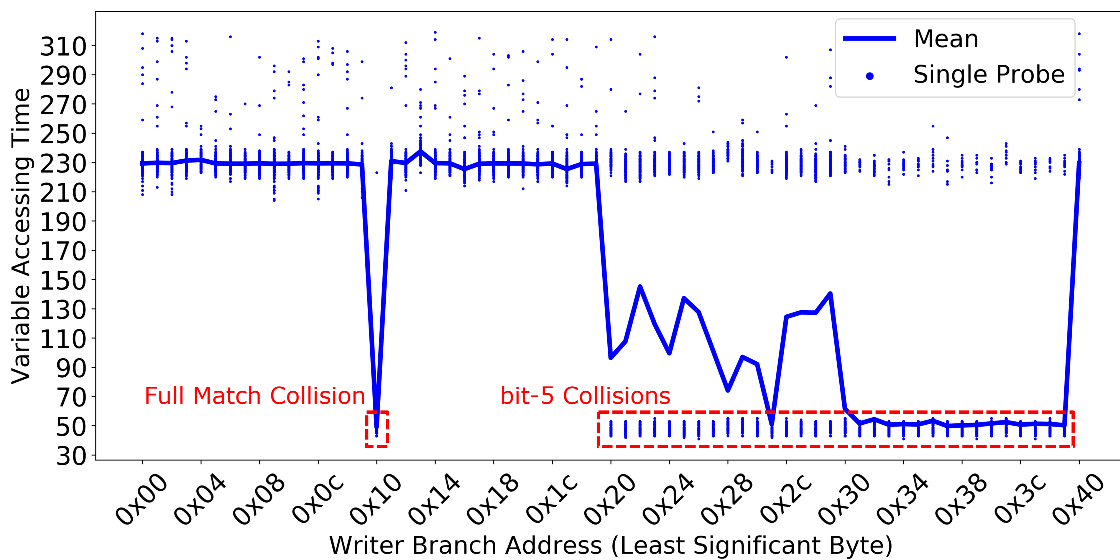


Figure 3.6: Demonstration of the two collisions types

3.2.3.2 Constructing a Portable Trojan

Trojans based on the early front-end branch collisions can achieve great covertness and portability. This is mainly due to two reasons. First, they do not rely on placing branch instructions far away from each other, contributing to their small size. Second, they do not rely on fixed addresses (aside from offsets within the cache line). This permits them to function when ASLR is enabled. In this type of trojans, all attack components (WB, RB, and the transient gadget) are encapsulated in a small chunk of code that fits into one or few cache lines. As a result, a malicious developer can prepare a portable block of normal C/C++ code that when compiled will act as a trojan. Such trojan will function as expected even if compiler reorders the functions inside binary or the executable is run with ASLR. This opens new vectors for spreading transient trojans. Instead of standalone applications, they can be distributed as shared libraries, patches, or via multi-party software development projects. The requirement for code to be aligned within a 64-byte block is possible to fulfill using various code optimization techniques such as function attributes [48] available in most compilers.

We demonstrate the functionality of the bit-5 collision by creating a simple trojan consisting of two functions, `f1` and `f2`. The high-level overview is presented in Figure 3.7. We assume `f1` is a function that has access to sensitive data. For instance, this can happen when `f1` is a secret key manipulation function, and the key is loaded in one of the architectural registers in function's prologue (for example `%rax`). In addition, `f1` contains an indirect branch instruction. This can happen because of a `switch()` statement or a call to a virtual method. The code in `f1` does not contain any instructions capable of leaking secret data via covert channels. It assumed that this function will be inspected for that matter. Another function `f2` is a not-sensitive function that is located directly below `f1` in virtual memory, permitting the bit-5 poisoning. Since `f2` does not contain any memory accesses to sensitive data, the presence of a transient gadget in its body does not violate security properties and will not be flagged as dangerous code during analysis. However,

due to the branch collision, `f2`'s function body will be executed (in transient mode) in the context of `f1`. By context here we understand the data accessible by each function. This enables a unique transient execution attack. *Due to colliding branches, the architectural state is violated in such a way that results in the body of one function to execute with the context (data) of another function.* For demonstration, we utilize a gadget similar to the gadgets used in prior work [79, 150, 77, 33]. The gadget reads the secret byte and then reveals its value by initiating a memory access using the address dependent on that value.

To evaluate the effectiveness of this type of trojans, we performed an experiment with the code illustrated in Figure 3.7. We first execute function `f2`, which moves the non-secret value 256 into the register `%rax`. Then it executes the WB, which transfers execution to the gadget that outputs the value stored in the register via leaving a trace in cache. Next, we execute function `f1`, which places the secret value 42 into the same register. Only `f1` has access to that value. The function then activates the RB, resulting in transient execution jumping to the body of `f2`, which contains the gadget leaking the value stored in register `%rax`. Please note, when the gadget instruction is executed in transient mode, the register contains the secret value. After both functions are executed, we probe the cache covert channel by checking all possible byte values transmitted by the gadget (from 0 to 255). If no cache hits are observed, we record no byte transfer. If a transferred value is detected other than 42, we detect an error. Otherwise, we register a correctly transmitted bit. In a real-world trojan, capturing leaked bits is typically performed in another process, or it may affect the timing of an externally observable event. However, for simplicity, we place all components into a single process. In addition, to insure RB's misprediction, we flush the correct target from cache on every iteration. We configured our PoC to leak 10 kilobytes of data and ran it 10 times. The average number of iterations required to transfer 1 byte was 43.69, and the average error rate was 0.0450%. The large number of iterations indicate that bit-5 poisoning does not happen frequently. In Section 3.3, we present an automated approach to optimizing such trojans allowing to improve their throughput significantly.

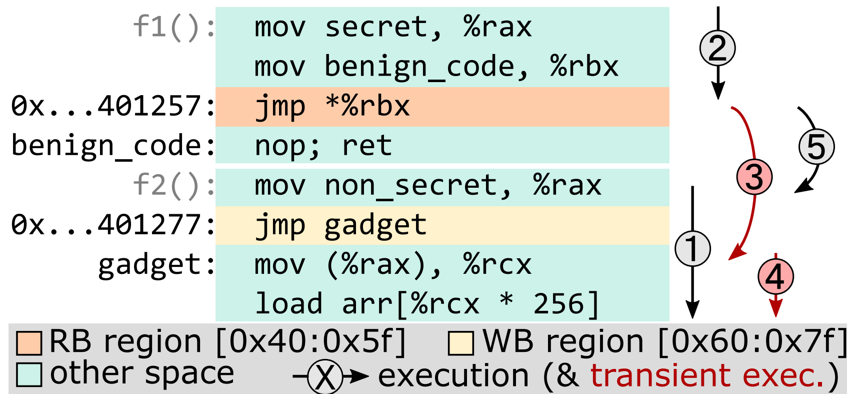


Figure 3.7: Portable transient trojan example

3.2.3.3 Dispersing Gadgets to Avoid Detection

Transient execution attacks rely on gadgets to leak sensitive data. Recently, several works proposed detecting these gadgets [54, 151, 3, 5, 33]. They are largely based on performing static binary analysis. To bypass such detection, we developed a technique based on the newly discovered collision pattern. Static analysis tools rely on detecting code sequences that result in the following actions: 1) memory location is read, and 2) another memory access is performed with an address dependent on the value of the first operation. These solutions use abstract interpretation of binary code to find data dependencies and match activities with known malicious patterns. They are effective in detecting gadgets even if the attacker tries to obfuscate them by using different variables and registers. However, abstract code interpretation does not account for side effects of transient control flow transition due to a bit-5 collision. We can utilize this anomaly to violate the architectural state and disperse a transient gadget into two parts, each of which is not identified as a malicious instruction sequence. Figure 3.8 shows a gadget consisting of 4 operations. Following the described approach, we add an indirect jump instruction and refactor the code in such a way that the first two operations are executed before the poisoned jump and the last two after. From the architectural state point of view, the second part of the gadget will never be executed. However, due to the poisoning, the transient execution will result in full gadget execution. After this transformation, the code will produce exactly the same

transient execution effect. Since we are the first to report the bit-5 collision; we believe that this technique is capable of defeating solutions based on gadget detection.

To evaluate the effectiveness of this technique, we compared the number of iterations required to leak 10KB using the bit-5 based trojan with and without dispersing the gadget. To do that, we moved two of the gadget's instructions before the RB. The average number of iterations required to transfer 1 byte from 10 runs was 20.41, and the average error rate was 0.0147%. These results indicate that dispersed gadgets are roughly two times more efficient. This is due to reducing the number of gadget instructions that execute in transient mode by moving them before the indirect jump. Therefore such a technique can be used not only to avoid detection but also to improve the gadget performance.

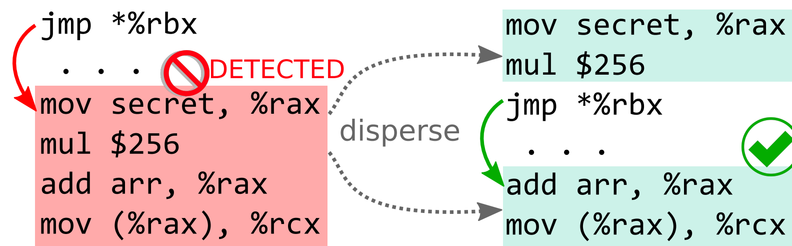


Figure 3.8: Dispersing a transient gadget to avoid gadget detection tools. Solid arrows indicate transient execution flow

3.2.4 Skipping Branch Trojans

3.2.4.1 Skipping indirect branches

In addition to collisions between different branches, CPUs we tested based on AMD Ryzen and Intel Haswell architectures have another indirect branch-related anomaly that can be used to construct trojans. In particular, when a prediction is not available in BTB, the CPU simply skips the indirect unconditional branch instruction and proceeds to the following instructions. In addition to constructing trojans, this mechanism can also be utilized to confuse static or dynamic analysis tools. Consider a program in which a certain function is invoked using an indirect call instruction. Assume its target is set during the program initialization and never changes. A detection tool will be able to find this correlation and

mark the program safe. However, due to indirect call skipping, a temporal architecture state violation will take place. Intel documentation confirms that indirect branches may be predicted non-taken [6].

3.2.4.2 Skipping based transient execution attack

The indirect branch skipping mechanism can be utilized to construct trojans with unique properties as they do not rely on known elements of previous Spectre-related attacks. In particular, they do not require conditional branches as in Spectre v1 or branch collisions as in Spectre v2 to violate architectural state.

To demonstrate the practicality of this approach, we designed a simple trojan application based on this mechanism and complied it using llvm. Figure 3.9 demonstrates its code with the disassembly of the key elements. Two functions are called via function pointers, and such calls are compiled to indirect call instructions. Function pointer `f1` is used to call the function that returns a secret value, which is then loaded into variable `sec`. The function pointed by `f2` loads a non-secret value into `nonsec`. After these two function calls, a gadget code sequence reveals the value of `nonsec`. Since its value is not secret, it is not considered a violation. According to System V ABI, functions are required to return the values using `%eax` (or `%rax`) register. After the return, caller function stores `%eax`'s value as a local variable on stack.

In the example code, the violation of architectural state happens when function call `f1` is **not** skipped while `f2` is skipped. This results in code ❶ loading the secret value into register `%eax`, followed by saving it in `sec` and then immediately transmitting execution to code ❷, which stores `%eax`'s value in `nonsec`. As a result, both variables temporarily hold exactly the same secret value. Then the gadget successfully reveals the value of the secret data via the cache. Please note that to enable the condition when one function is skipped while another is not, pointers `f1` and `f2` must be located in different cache lines. This can be done by adding or removing local variables in the parent function. For this experiment, we flush `f2` from cache. In a real-world attack, this can be done by finding an

eviction set [148].

To evaluate this trojan's accuracy, we executed it on an AMD Ryzen machine leaking 1KB and ran it 10 times. The average number of vulnerable function activations required to leak 1 byte of data was 888.07 with average error rate of 1.74%. Such a relatively low success rate can be explained by the attack relying on an infrequent event when one function is correctly predicted while another is mispredicted. The success rate can be further improved by manipulating with BPU prediction mechanism.

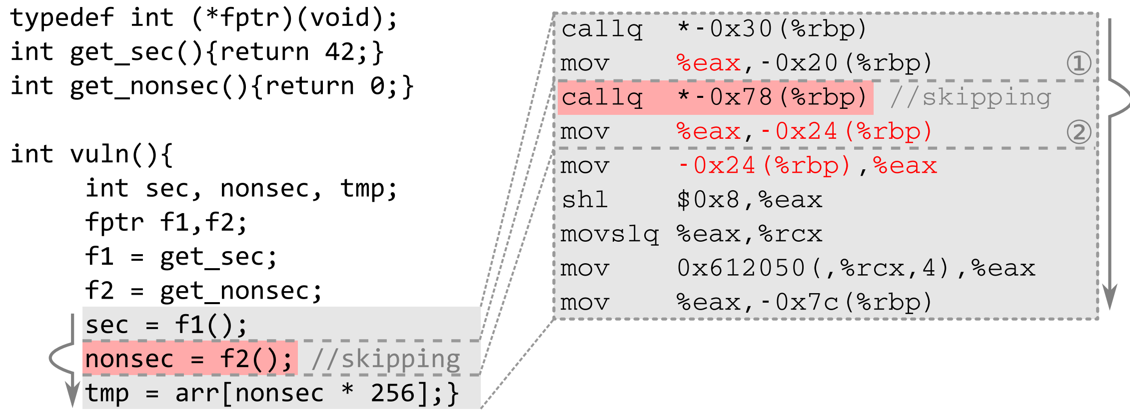


Figure 3.9: Transient trojan based on branch skipping

3.3 Improving Trojan Activation Rate

Effectiveness of transient trojans can be measured by their successful activation rate, which is the percentage of cases when data is leaked compared to total activation attempts. In our initial trojan implementation, the rate appears rather small, for instance, 12.79% and 4.86% for kernel and DSB based trojans, respectively. We noticed that trojans are sensitive to their surrounding code, which can either increase or decrease the success rate. This effect is especially noticeable for portable trojans since they are based on tight race conditions within the CPU front-end. Surrounding code, the code that is executed right before or immediately after the trojan's critical parts can cause various effects (both positive and negative). For instance, it can flush out buffers such as DSB, load store buffer, instruction cache and introduce contention in decoders, functional units, and ports.

Manually tuning trojans for these microarchitecture events is a difficult and meticulous task. First of all, many of the front-end components are not completely reverse engineered. Secondly, fine-tuning one property may affect other properties in a non-trivial way resulting in success rate degradation. Instead of reverse engineering and manual fine-tuning, we propose a method based on genetic programming that enables automatic trojan optimization based on injecting lightweight code artifacts. These artifacts serve no purpose other than creating various microarchitecture conditions and do not affect program's architectural state. Our method is shown to be effective, improving our initial portable trojan implementation from 4.86% to 98.35% resulting in the leakage rate of 13.5 kilobytes per second.

In the first stage of our genetic algorithm approach, we transfer a trojan into a mutation template. This template includes all elements of the original program with additional anchors, places in source code where random activities will be added in the future. The anchors are placed in locations that are likely to interfere with key elements of the trojan, for instance, adjacent to WB and RB. We discovered that trojan accuracy could be affected by adding blocks of `nop` instructions, which affect the code alignment and empty loops that load CPU resources handling branches. For our initial experiment, we used the portable trojan from Section 3.2.3.2. We placed a total of 15 anchors: 9 `nop` anchors and 6 loop anchors. The `nop` anchors inject 0–150 `nop` instructions while each loop anchor injects a loop with 0–8000 iterations. This results in 10^{43} possible combinations making the brute-force approach not feasible.

Instead, we perform the optimization by starting from 100 initial candidate solutions. We do so by randomly selecting values for each anchor. Then we use a simple genetic algorithm to find an optimal configuration. We set our initial fitness threshold (trojan success rate) at 20%. In each round, we apply an objective fitness function to each candidate, removing all candidates that have an attack rate lower than the fitness threshold. Then we sort the remaining by fitness score. A generator function performing crossover and mutation is applied to a subset of the remaining candidates with the highest fitness scores to create a new variation population of 100 candidates. During this phase, we apply a simple

heuristic to avoid crossover between very similar candidates ensuring that we continue to have population diversity in each round. This also reduces the risk that our algorithm converges to a suboptimal solution. We also guarantee 20% of each population to be entirely random to increase population diversity.

We compare the genetic programming approach to a simple random-based optimization. Here instead of performing mutation, we keep generating random candidates and select the best performing candidate in each round. Both approaches tested 2 000 trojans in 20 groups, 15 times, and their best 5 runs are demonstrated in Figure 3.10. The result shows the maximum trojan attack rate only incrementing when a more optimized trojan is found. The genetic algorithm converges to a trojan configurations that produce 90%+ attack rates, finding trojans with high attack rates quicker and 30% higher than the randomization-based approach. That highlights the benefits of using genetic algorithms for optimizing attacks based on microarchitectural effects.

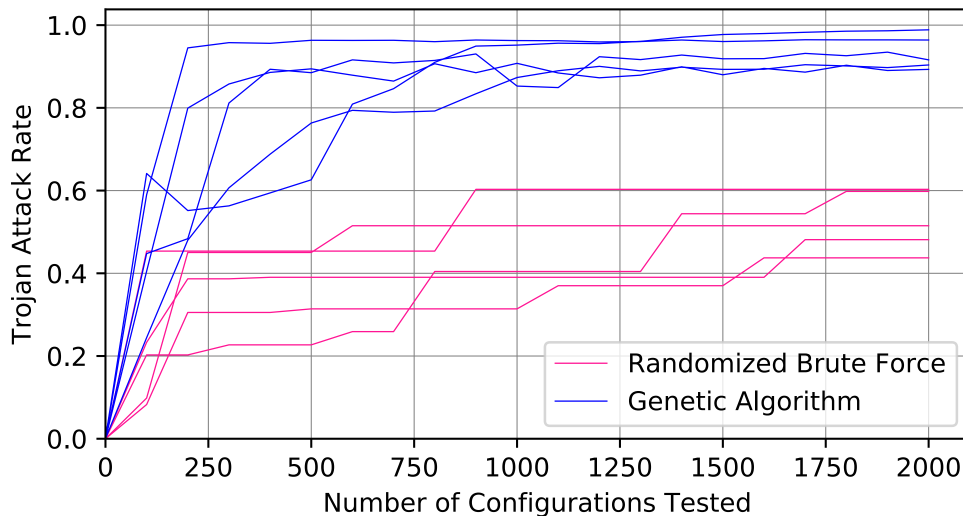


Figure 3.10: Genetic and randomization optimizer comparison

3.4 Detecting Collisions in Existing Binaries

Branch instruction collisions can occur naturally in regular executables. A typical binary on average contains one direct branch instruction per 4–7 instructions making collisions

between indirect and direct branches a common event. An advanced attacker may construct a trojan utilizing these collisions. In this section, we evaluate such naturally occurring collisions in existing binaries and reason about their use in attacks. For our analysis, we use Skylake architecture as a reference. We group all collisions in two types: portable and non-portable. The portable collisions are based on bit-5 mechanism, and their functionality is not tied to hard-coded addresses. Thus they function even in the presence of ASLR, unlike the non-portable collisions, which are based on the distant collision mechanism from Section 3.2.2. Each executable is analyzed in its normal running context to detect collisions between branches in executable and its libraries.

We developed a light-weight binary analysis tool to find locations where WBs and RBs produce portable and non-portable collisions. First, each binary is disassembled, then we perform a search for all direct and indirect branch instructions. All potential WB and RB instructions are then passed to a BTB mapping function, which is based on Skylake BTB reverse engineering to find their index, tag, and offset bits. Our tool then identifies WB-RB pairs that collide according to two types of collisions.

Figure 3.11 demonstrates the results gathered from processing 16,015 binaries native to Ubuntu 18.04, including user applications, libraries, and kernel modules. The X-axis shows total indirect branches in executable, while the Y-axis all possible collisions, including collisions between library and code segments. Please note that since distant same address space collisions are sensitive to ASLR, there will be different sets of collisions appearing each time the program is rerun. Although at first this may appear as a negative effect, an advanced attacker can use this phenomenon to further hide a malicious trojan by making it activated only under certain ASLR bits. This makes the analysis of all potential collisions and their effects infeasible. To give a high-level overview of the number of such collisions, we perform the analysis with ASLR deactivated. At the same time, the DSB collisions are not sensitive to ASLR. As seen from the result, existing binaries contain large numbers of naturally occurring collisions of both types. The collisions tend to linearly grow with the total count of indirect branches present in a given binary. For example, Google Chrome

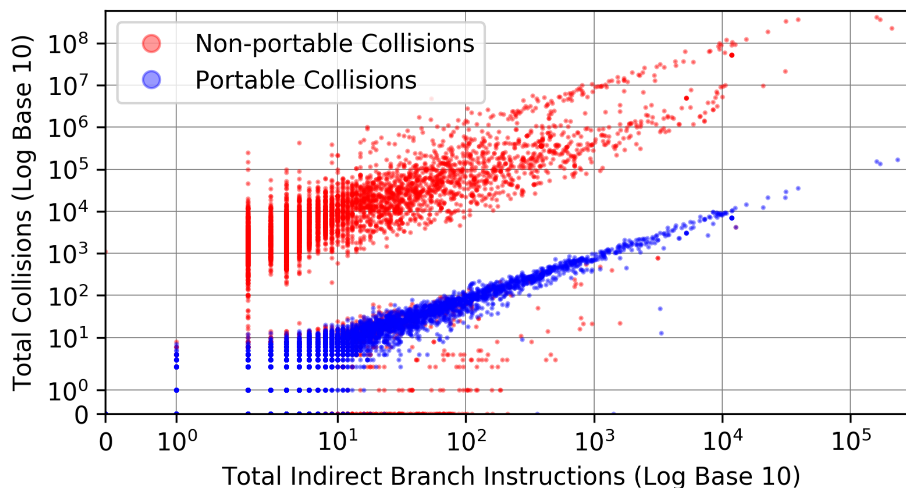


Figure 3.11: Analysis of branch collisions in existing binaries

executable contains a total of 170k indirect branches resulting in 136k portable and over 300 million non-portable collisions. Such a high number makes hiding malicious branches a relatively easy task as the analysis of all potential transient execution effects becomes very difficult.

As we discussed in Section 3.2.4.1, indirect branch instructions can violate architectural state even when no collisions are present. Thus, every single indirect call and jump instruction (X-axis in Figure 3.11) has the potential of doing so. As a result, we believe any indirect branch should be treated as a potential security threat unless CPU design can ensure that transient execution can never leak sensitive data.

3.5 Countermeasures

Since indirect branch instructions are required for our attacks to function, retpoline sequences can be used as effective mitigation. However, retpolines must be added during compilation and cannot be applied to precompiled binaries. Because retpolines lead to code bloating and performance overhead [16], current binaries seldom use this technique.

Distant same address space branch collisions can be prevented if future BTB designs store full addresses (e.g., tag and target) instead of their reduced or compressed versions. However, such a design would significantly increase the BTB size and, therefore, costs of

hardware.

Mitigating bit-5 collisions in hardware appears a more challenging task since it would require a front-end redesign. For instance, a naïve solution is to delay BPU predictions until instruction boundaries are determined. However, that would lead to introducing delays when processing branch intensive μop sequences from DSB. Alternatively, a software-based solution can be developed to sufficiently space direct and indirect branches with binary editing at runtime or by manipulating compiler code generation primitives to prevent placing direct and indirect branches in the same 64-byte block. However, that would lead to significant code bloating. In addition, our collision detection tool can be used to find potentially dangerous branches and inject in-place mitigations such as `lfence` instructions. Future microarchitecture designs are urged to adopt better mechanisms that do not permit branch instruction anomalies, for instance, by adding a type field in the BTB to prevent direct and indirect branch collisions and avoiding indirect branch skipping. A recent work by Yu et al. [163] proposed a light-weight hardware solution based on preventing unsafe data accesses being forwarded to transient execution.

3.6 Related Work

To the best of our knowledge, our work is the first work analyzing the security effects of branch collisions within same address spaces. In addition, we introduced a new type of malicious software that utilizes transient execution in the form of self-contained transient trojans represent.

Wampler et al. successfully created a malware program with a transient execution payload [150]. However, malicious software modules presented in their work require a separate activation process. Moreover, a correctly configured IBPB would force BTB flushing on context switched, making poisoning across different processes impossible. All types of our trojans work with current microcode-based protections enabled. Kiriansky and Waldspurger developed Spectre 1.1, where transient buffer overflows can be used to jump transient execution into arbitrary code. This Spectre buffer overflow attack can be used to

redirect execution to instructions after a serialized instruction (Spectre V1 mitigation) [77]. Canella et al. performed an analysis of 12 Spectre variants, including the possibility of multiple same address space Spectre attacks [31]. However, the analysis did not reason on how these vectors can be utilized to construct practical exploits.

Recent works have been published regarding the detection and mitigation of Spectre attacks. SPECTECTOR by Guarnieri et al. detects transient information flows [54], and the principles behind this work can be applicable to the detection of transient trojans. However, without a completely accurate collision model, this and similar tools may overlook dangerous transient execution flows presented in this work. Our work makes a contribution by expanding upon the existing collision model. Finally, Depoix et al. developed a method of detecting Spectre attacks by identifying Spectre attacks using machine learning [38].

3.7 Conclusions

In this work, we presented a new type of practical attack based on transient execution. We demonstrated transient trojans — malicious software modules that utilize BPU anomalies happening inside software entities. In addition, we reverse-engineered the BPU addressing scheme, which allowed us to detect new exploration mechanisms. Utilizing them, we were able to create trojans that have several properties desirable for attackers such as being portable, working in the presence of any microcode-based protection mechanisms, and the ability to stay undetected by current detection tools. We believe our work improves the current understanding of attacks based on transient execution by bridging the gap between exploitable hardware primitives and constructing realistic attacks. This work has been summarized and published in ASPLOS 2020 [164].

Chapter 4

STBPU: A Reasonably Safe Branch Predictor Unit

4.1 Introduction

Over the last few decades, a large number of protection techniques against software attacks have been introduced making exploitation of traditional attack vectors such as code injection or return-oriented programming challenging. With a decreasing number of available targets for software attacks, the attention of adversaries is more frequently drawn to exploitable weaknesses in hardware. Although hardware attacks such as microarchitectural side channels [21, 110, 108, 9, 63, 11, 49, 91], covert channels [43, 61, 106, 99], and power analysis [80, 101, 97, 10, 107] attacks have been known for a long time, only recently did researchers demonstrate the true power of microarchitectural attacks with newly discovered speculative execution attacks, such as Meltdown [89, 143] and Spectre [79, 33, 95, 76, 81, 128]. These attacks are based on speculative (transient) execution, a performance optimization technique present in nearly all of today's processors. While this technique improving CPU performance, with a carefully crafted exploit, it completely undermines memory protection, giving unauthorized users the ability to read arbitrary memory [79, 89] and disable crucial protections [76].

Microarchitectural attacks are possible because performance optimizations such as caches, prefetchers, and various predictors were not traditionally designed with security in mind. For example, data structures used to implement these mechanisms are commonly shared, making various conflicts possible. Some conflicts result in leakage of sensitive data. One such mechanism is the branch predictor unit (BPU). Substructures within the BPU are typically shared, and the stored data is compressed, prone to various branch collisions. [40, 119]. This enables attacks such as side channels [10, 44, 42] that are capable of leaking encryption keys or bypassing ASLR, and the recently introduced speculative execution attacks [79, 76]. At the same time, shared BPUs are beneficial for performance. They allow high utilization of hardware structures to reduce the cost and enable efficient branch history accumulation. [103]. Therefore, naïve protections which disable sharing or flush BPU structures have high performance overhead. Recent Intel microcode updates introduced as a countermeasure against Spectre attacks [64] demonstrated that the overhead from naiv̈e protections can be as high as 440% [137, 113].

Despite significant efforts directed towards designing other secure microarchitectural components e.g., caches [152, 74, 154, 118, 37, 147, 90] and memory buses [126, 12, 87], secure BPU designs remain a handful of attempts [85, 52, 149]. More importantly, none of existing approaches completely eliminate BPU vulnerabilities. such as recent μop caches attacks [123]. We propose Secret-Token Branch Prediction Unit (STBPU), a safe BPU design aimed to protect against collision-based BPU attacks and eliminate BPU side channels.

STBPU prevents collision-based side channel and speculative execution attacks by disallowing software entities from creating controlled branch instruction collisions and thus affecting each other in an unsafe way. This is done by customizing the branch representation for each software entity in the form of address mappings and by encrypting data stored in BPU. In STBPU, each software entity is provided with a unique, randomly-generated secret token (ST) that customizes data representations. STBPU constantly monitors active attacks with hardware events and re-randomizes STs to prevent attackers from reverse-engineering

the ST tokens.

This work makes the following contributions:

- We propose, STBPU, a safe BPU design protects against speculative execution attacks including fast algorithm attacks, provides strong isolation guarantees to eliminate to BPU side channels, and incurs low overheads.
- We provide insight on how to adapt the STBPU design, invalidating the recent μop cache attacks which can bypass mitigations e.g., invisible speculation and fencing.
- We design an automated frame to create lightweight ST-dependent remapping functions and validate STBPU with in-depth security analysis .
- We evaluate STBPU performance on advanced prediction mechanisms e.g., TAGE_SC_L and Perceptron and show low overhead even with extreme security settings.

4.2 Threat Model

We assume a powerful attacker that has a *complete* understanding of all hardware components and structures in the STBPU. The attacker has access to normal reverse engineering resources, such as time measurements and performance counters, and has access to a wide variety of hardware covert channels. The STBPU design calls for new special purpose registers as detailed in Section 4.3; the adversary is assumed to be unable to read/modify the contents of these registers. Such a role is delegated to a privileged software entity (OS, hypervisor) which attacker does not control.

We assume the attacker cannot gain access to ST for the victim process neither when it is in the special purpose register, nor when system software stores it. Former is impossible because the register can only be accessed from the privileged mode. Later happens only in the event of system software compromise. The ST can be considered as part of processes's context that is saved and restored on context switches. The event of attacker gaining access to context data would be equal to a full compromise. In such case, there is no point for attacker to use side channels.

We consider attacks presented in Table 2.1 including both side channel attacks in which

victim executes a sensitive data dependent branch branch as well as speculative execution attacks where victim is forced to speculatively execute leakage gadget code. We assume the following two attack scenarios:

Sensitive Process as Victim. In this scenario, an attacker tries to learn sensitive data from a victim process by manipulating the BPU state and recording observations. The attacker has control over user-level process co-located on the same CPU core and is capable of performing activities that are normally allowed to untrusted process such as accessing fine grain hardware timers via `rdtscp` instructions. We assume the victim and attacker can either execute on two logical cores within the same physical core or share the same logical core with time-slicing. This scenario also includes recently introduced transient trojans [164] where collisions occurring within the same memory segments are exploited.

Kernel/VMM as Victim. The attacker takes a form of a software entity with lower privilege level, i.e. untrusted user process. The attacker tries to learn sensitive data owned by a higher privileged entity (OS kernel or VMM) by manipulating with BPU state and recording observations. Here, victim and attacker share a same contiguous virtual address space. Attacker is restricted from executing privilege instructions.

4.3 STBPU Design

As discussed in Section 2.6, BPU attacks are possible due to deterministic mapping mechanisms, allowing attackers to create branch collisions. STBPU aims to stop these attacks by replacing these deterministic mechanisms with keyed remapping mechanisms which prevent branch collision construction. The design philosophy of STBPU is to create different data representations for separate software entities inside the BPU data structures. Each software entity requiring isolation is assigned a unique secret token (ST), which is a random integer that controls how branch virtual addresses are mapped into BPU structures. This ST is also used to encrypt/decrypt stored data. Compared to naïve protections based on flushing or partitioning, our approach has a number of benefits.

Consider a protection scheme where branch target poisoning is prevented by flushing

the BTB on context switches. Invalidating the entire branch target history will negatively affect performance in cases where context switches are frequent. Similarly, to protect from target collisions between kernel and user branches, BTB must be flushed on mode switches (e.g. all syscalls). Partitioning hardware resources reduces the effective capacity of BPU structures resulting in a higher miss rate and lower prediction accuracy. Instead, a customized mapping approach allows separate software entities to co-exist in the BPU with minimal performance overhead; performance evaluated in Section 4.6. STBPU utilizes two key approaches to enable safe resource sharing.

- STBPU makes collision creation difficult by ensuring all remapping functions are dependent upon both branch address and ST.
- STBPU detects when a potential attacker process has recovered sufficient information that enables deterministic collision creation by monitoring hardware events.

4.3.1 ST re-randomization

The ST of the current process in the BPU is re-randomized once certain (OS controlled) thresholds are reached. Note that in STBPU design, the OS is trusted and is responsible for setting parameters such as the re-randomization threshold. This is a common assumption for systems protecting against microarchitectural attacks since compromising OS gives the attacker full control over the system, making such attacks non-necessary. On the other hand, such a design choice makes our mechanism more flexible and permits the OS to adjust the strength of enforcement based on factors such as whether a certain process is considered sensitive or the attacker's capabilities. For instance, if a more effective side channel attack is discovered after STBPU is deployed, the underlying hardware mechanism will still remain effective and will only require the OS to readjust the thresholds. Moreover, for the extreme cases of sensitive processes the OS may opt to set the threshold as low as 1, forcing re-randomization after every branch instruction, effectively disabling the BPU mechanism.

STBPU can be also adapted for systems with OS not trusted (e.g. SGX), then another

system component needs to be responsible for managing tokens and thresholds. For instance, in the case of SGX, the enclave entering routine can serve this purpose. Alternatively, simple logic of ST management in STBPU should also enable hardware only implementation. Re-randomizing ST effectively resets the customization of the BPU data representation for that process. Although it leads to the loss of branch history (by making it unusable), our analysis indicates that such events are infrequent. Re-randomizing the ST of one process does not remove stored history of a process with a different ST. This is the key difference compared to flushing-based approaches. We derive the re-randomization thresholds through the analysis in Section 4.5.

While potentially dangerous, branch history sharing between programs benefits performance. Consider a server application that spawns a new process for each incoming connection. Since each process executes the same code, the accumulated BPU state is used by the newly spawned process. This allows the new process to avoid the lengthy period of BPU training. STBPU permits selective history sharing by allowing OS to provide multiple copies of the same program to utilize the same ST value. However, when sharing is not desired, each thread can be given a unique ST.

4.3.2 Hardware Mechanisms and Interfaces

Since current BPU designs are highly optimized in terms of performance and hardware cost, we restrict ourselves to only modifying BPU mapping mechanisms, adding registers, and encrypting stored targets. Such changes will provide similar performance to the unprotected design and make STBPU agnostic to a particular BPU design. In STBPU, each hardware thread is provided with an extra register to store the ST of the current process. Only the OS is allowed to read/modify these registers, and these registers are inaccessible in unprivileged CPU mode. As such, the OS facilitates history retention across context/mode switches by loading the appropriate STs. We also add several model-specific registers (MSRs) that store thresholds and counters for automatic ST re-randomization. These MSRs monitor the events that indicate an active attacker process. We monitor two events: i) branch

	Baseline input	STBPU input	Output	Function
1	32 s	32 ψ, 48 s	9 `ind`, 8 `tag`, 5 `offs`	$R_1(80 \mapsto 22)$
2	58 `BHB`	32 ψ, 58 `BHB`	8 `tag`	$R_2(90 \mapsto 8)$
3	32 s	32 ψ, 48 s	14 `ind`	$R_3(80 \mapsto 14)$
4	18 `GHR`, 32 s	32 ψ, 16 `GHR`, 48 s	14 `ind`	$R_4(96 \mapsto 14)$
t	48 s, `L(GHR)`	32 ψ, 48 s, `L(GHR)`	10/13 `ind`, 8/12 `tag`	$R_t(80\uparrow \mapsto 25)$
p	48 s	32 ψ, 48 s	10 `ind`	$R_t(80 \mapsto 10)$

`L(GHR)` — represents geometric series of global history lengths
`s` — represents the source bits of branch instructions

Table 4.1: I/O bits for baseline and STBPU functions

mispredictions which includes incorrectly predicted direction of conditional branches and targets of any branch, and ii) BTB evictions. In Section 4.5, we explain how these events are utilized to deter BPU attacks. Initially, the counter values are set to their respective threshold values. When an event is observed, the corresponding counter is decremented. When a counter reaches 0, the current ST is re-randomized, and the CPU reset the counter with the threshold value. The OS treats these registers as a part of software context saving, and recovering their values on context/mode switches. We assume re-randomization is done by fetching a value from low-latency in-chip pseudo-random number generator [92].

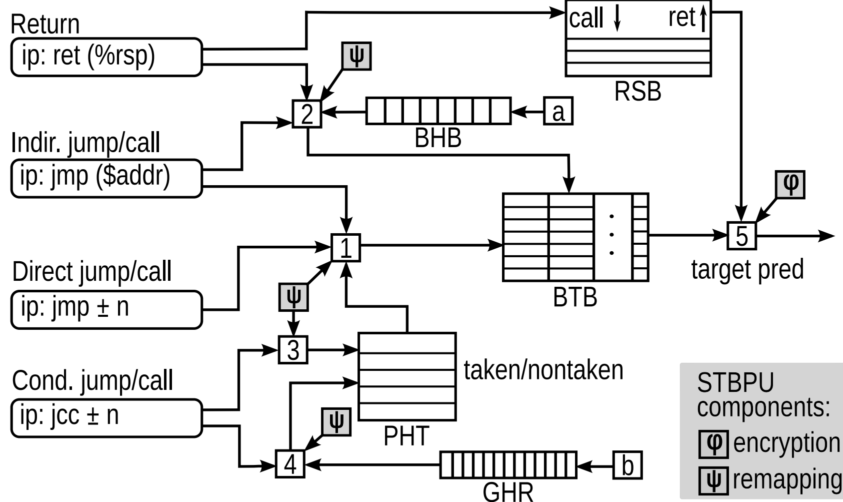


Figure 4.1: BPU with STBPU components highlighted

The ST register is a 64-bit register divided into two 32-bit chunks, ψ and φ. The first chunk ψ acts as a key for a keyed remapping functions making BPU mapping unique for each process. Figure 4.1 demonstrates the STBPU design on top of the components of

a baseline branch predictor depicted in Figure 2.1. In STBPU, we replace functions 1, 2, 3 and 4 with STBPU remappings $R_{1..4}$ accordingly. We add functions R_t and R_p that are used for STBPU implementation with the TAGE and Perceptron predictors. Both baseline and STBPU remapping functions reduce input data (address, BTB, GHR bits) into fixed size index, tag, and offsets used by the BPU to perform lookups. Section 4.4.1 describes how $R_{1..4,t,p}$ were selected. Additionally, these functions utilize the entire 48-bit virtual address unlike legacy functions that use truncated address bits as inputs. This is crucial to prevent the same address space attacks [164]. Table 4.1 details all input/output bit changes between the baseline and STBPU models.

We use a simple scheme based on XOR to encrypt data stored in BPU structures to stop attackers from redirecting execution to a desired speculative gadget even if collisions occur. In the case of a collision, speculative execution will be redirected to an encrypted (random) address. This will effectively stall malicious speculative execution. In STBPU, every entry stored in BTB and RSB is XORed with φ of the current process. Note that the baseline BPU stores only 32 bits of target addresses, so the 32-bit φ is sufficient for encrypting all stored bits. We use a simple XOR encryption for two reasons: i) XOR operations are extremely fast with trivial hardware implementation, and ii) automated ST re-randomization makes the simple XOR encryption sufficiently strong (discussed in Section 4.5). To decrypt data in BTB and RSB, we modify the function 5, which XORs target bits with φ before extending them to 48-bit address.

4.4 Implementation

In Section 4.3, we defined remapping functions $R_{1..4,t,p}$ which replace the methods of calculating indices, tags, and offsets for lookup purposes in the baseline BPU model. Remapping functions $R_{1..4,t,p}$ can be thought of as *non-cryptographic* hash functions. Given the size constraints of the BPU structures, collisions between different inputs to functions $R_{1..4,t,p}$ will occur; this fact prevents functions $R_{1..4,t,p}$ from providing cryptographic security, regardless of implementation. This inherent weakness is remedied with periodic re-randomization

of STs; the security of such re-randomizations are discussed in Section 4.5. The mapping functions used in the baseline model are not fully reverse engineered, but we can safely assume some fast compression functions are used with delays of no more than 1 clock cycle. Using performance and security as our guides, we placed several important constraints upon functions $R_{1..4,t,p}$:

- C1 The compute delay for $R_{1..4,t,p}$ must not exceed C clock cycles, where C may vary from CPU to CPU. For our purposes, we choose C to be 1 clock cycle. We enforce this by limiting the number for transistors of each remapping function on the critical path.
- C2 The function must provide *uniformity*: outputs of $R_{1..4,t,p}$ should be uniformly distributed across their respective output spaces.
- C3 The function must demonstrate *avalanche effect* [60]: The outputs of $R_{1..4,t,p}$ must appear to be pseudo-random, and the relationship between inputs and outputs should be non-linear.

We analyzed existing hardware supported hashing mechanisms, but found none that satisfied our specific requirements. Specifically, existing multi-round hash functions exceed the single CPU cycle constraint. Later we describe a mechanism we developed to automatically generate remapping functions taking into account aforementioned constraints. In addition to remapping, STBPU requires encryption of branch addresses stored inside BPU. We found out that existing lightweight cryptographic functions are not suitable for our purposes for two main reasons: First, using strong ciphers does not directly translate into better security which are primarily designed to withstand known plaintext/ciphertext attacks. However, STBPU threat model is much different as attackers never observe encrypted addresses (ciphertext) nor partially matched plaintext/ciphertext. They only observe collisions (not knowing with their own or victim's branch) and need to reverse-engineer the rest of the address bits. Besides, knowing their own STs does not provide immediate access to collision creation or simplifies collision-based attacks. In Section 4.5, we show that the number of mispredictions and evictions attackers must incur to successfully infer a ST far exceeds the

thresholds that will trigger ST re-randomization. Thus, encrypting with a more advanced cipher would not increase the level of security. Secondly, more sophisticated encryption schemes introduce significant delays in CPU frontend. For instance, we explored PRINCE-64 [27] and Feistel-Network [100] to encrypt stored branch targets. While comparably fast, PRINCE-64 and Feistel-Network will still consume multiple clock cycles and consume more energy due to higher number of gates compared to a simple subsingle-cycle XOR operation.

4.4.1 Automation of Finding Remapping Functions

Automated Remap Generation Algorithm. Designing the remapping mechanisms is a multi-variable optimization problem. To solve it, we developed an algorithm that takes in a list of hardware constraints, and randomly generates remapping function candidates. The algorithm composes the function from a predetermined pool of primitives. Each remapping function is iteratively generated and tested one layer at a time, where a layer is a block of these primitives. After a layer is added, the current function is tested against the supplied constraints. There are three possible scenarios that occur during each round of testing. i) The current design satisfies all constraints, and subsequently stored for later optimization. ii) The current design violates one or more constraints, and is discarded. iii) The current design does not outright violate the constraints, but is incomplete. In case 3, our algorithm changes the weights used for primitive selection during the creation of the next layer to improve the current design.

Constraint Selection of C1. Our algorithm requires an input of several variable constraints for the generated remapping functions to satisfy C1. These constraints are: the maximum count of transistors along the critical path, the maximum number of transistors in parallel (breadth), the maximum number of total transistors for the design, the number of input and output pins, the maximum number of functional layers (blocks) the design can have, and the maximum number of wires an arbitrary wire can cross over.

Modern processors are designed to perform 15-20 gate operations in a single cycle [115], which translates to roughly 30-45 transistors along the critical path. The delay incurred

by each transistor in the critical path is relatively independent of the CPU clock cycle; therefore, the faster the CPU's clock cycle, the smaller the number of transistors that can be completed within 1 clock cycle. Therefore, we assume 45 is the absolute maximum number of transistors we allow in the critical path with preference set for shorter critical paths.

Primitive Selection. Much research has been conducted into cryptographic hash primitives [84, 26, 25, 167, 166] that provide building blocks for hash functions with strong properties. We leverage these primitives from SPONGENT [26] and PRESENT [25] hashes. Out of those S-boxes (establishing non-linearity by substations) are perhaps most critical. To increase the simplicity of remapping function generation, we separate primitives into two categories: non-invertible compression primitives and mixing primitives.

Non-invertible primitives tend to employ XOR logic gates to obfuscate the relationship between input and output. For many such primitives, multiple inputs generate the same, smaller output which makes reverse-engineering difficult. Combining multiple non-invertible layers increases complexity of attacks aiming to pair a known output to an unknown input. These primitives *compress* input size $|m|$ to an output size $|n|$ where $|m| > |n|$. Table 4.1 shows the disparity between the input and output sizes for $R_{1..4,t,p}$ functions, and indicates the need for optimized compression primitives. Mixing primitives are primarily used to introduce *non-linearity* to a hash design which makes deterministically changing the output by varying the input difficult. These primitives are primarily composed of $|m| \mapsto |m|$ sized S-boxes and P-boxes (performing permutations). Since the hardware complexity of S-boxes increases superlinearly with the size of $|m|$, we limit our S-boxes to a maximum of 4 input/outputs. These S-boxes can be implemented efficiently with combinatorial logic or transistor/diode matrices. P-boxes are constrained by the maximum wire crossover set for the algorithm.

Validation of *Uniformity* (C2) and *Avalanche Effect* (C3)

Remapping functions that satisfy the hardware constraints are then tested against constraints C2 and C3. We first employ the balls and bins analysis and compute the

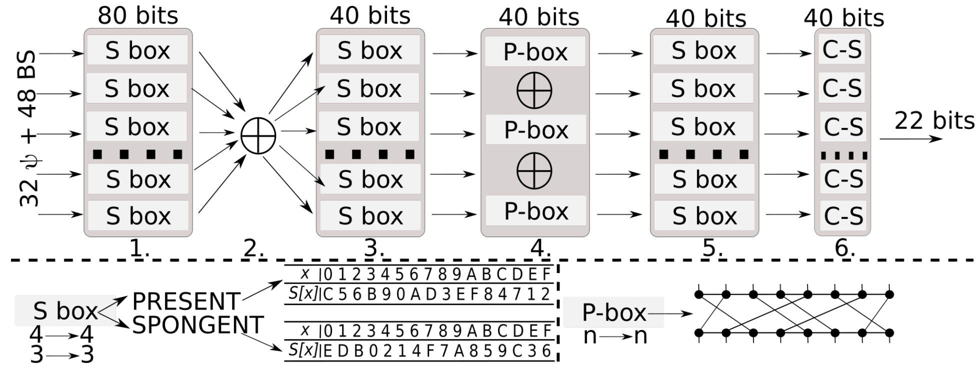


Figure 4.2: R_1 remapping function construction

coefficient of variation (CV) of bins to approximate the uniformity (C2) of the output space [117]. C3 is satisfied when a remapping adheres to a strict avalanche criterion. To quantify the avalanche effect of F, for each input λ, we generate a set of unique inputs, S, where each input in S differs from λ by a single bit flip. We then compute the hamming distance between $F(\lambda)$ and $F(S_i)$, for all inputs in S. Using these hamming distances, we determine the CV of the hamming distances for a particular λ. We test each F with 1 million random inputs and compute the average hamming distance for all inputs. The ideal case occurs when: i) the average hamming distance over 1 million random inputs is roughly 50%. ii) For all inputs, the CV of the average hamming distance for each input is 0. iii) For all bit positions of an output of F, the difference between the minimum and maximum hamming distances for a bit flip in any bit position is 0.

4.4.2 Optimization and Remapping Selection

The final selection of remapping functions $R_{1..4,t,p}$ is primarily based upon the results from the previous tests. The result is a multiobjective optimization problem where the ideal state for different desired metrics may be maximized or minimized. To make all metrics comparable, we normalized each metric so that the optimal value is 0. We then considered this to be a simple weighted optimization problem where we seek functions that yield the lowest sum of all metrics recorded when testing for uniformity and the avalanche effect. Let F be a particular function in the group of potential functions G for remapping function R_i,

for $i \in R_{1..4,t,p}$:

$$min \sum_{i}^{k} w_i g(F), F \in G \tag{4.1}$$

All weights were set to 1 to avoid prioritizing one metric over another. Further prioritizing then can be done by hardware developers for a specific CPU design. For space reasons, we do not show the designs for all of $R_{1..4,t,p}$ since they share many similar characteristics. Instead, we show the chosen design for R_1 in Figure 4.2 where stages 1, 3, and 5 are substitution layers using $4 \mapsto 4$ and $3 \mapsto 3$ S-boxes. For space reasons, not all types of S-boxes are shown. Under the design of R_1, we show the logical mappings for S-boxes used by PRESENT and SPONGENT. P-boxes are $n \mapsto n$ in size with the pin mappings generated randomly by our remap function generator. C-S boxes are compression structures that map $|m|$ bits to an output size of $|n|$ bits where $|m| > |n|$. This design of R_1 has a critical path length of 36 transistors, so it is capable of being computed within a single clock cycle.

4.5 Security Analysis

We assume any attackers can have complete knowledge of all STBPU remapping functions, full control of execution flow, and are capable of executing branches to/from any address within their processes. The goal is to enable malicious branch instruction collisions that allow mounting one of the collision-based attacks. STBPU makes collisions non-deterministic, forcing the attackers to rely on either brute force approaches or reverse-engineering the ST value. Further, attackers can utilize recently proposed fast attack algorithms such as GEM [116] and PPP [114] that target randomized caches [24, 28].

4.5.1 Analysis of Branch Predictor Attacks under STBPU

An attacker possessing knowledge of their ST (ψ/φ) voids the security provided by the STBPU because they can deterministically generate outputs with any of the remapping functions used by the STBPU. Before we discuss how STBPU affects attacks on BPU, we

Parameter: Description	A: Branch in attacker(A)'s address space
W_{struct}: Number of ways	V: Branch in victim(V)'s address space
I_{struct}: Number of sets (indexes)	$\psi_{a/v}$: A/V $R()$ 32-bit token
T_{struct}: Entry tag bit entropy	$\varphi_{a/v}$: A/V target encryption token
O_{struct}: Entry offset bit entropy	τ_Q: Target of arbitrary branch Q
Ω_{struct}: Entry target bit entropy Q	E_Q: Entry stored for arbitrary branch Q

Table 4.2: Parameters used in STBPU analysis

show the parameters for security analysis in Table 4.2 and list several important axioms below:

A1 Attackers do not know the numerical outputs of $R_{1..4,t,p}$.

A2 Due to A1, all the current state of the STBPU must come from detection of mispredictions and evictions.

A3 Attacker does not have inherent knowledge or control of ST of any process.

4.5.1.1 Target Injection Attacks

Recall that we encrypt the targets stored in the BTB and RSB through the following means: $E_A = \varphi_a \oplus \tau_A$. With Spectre V2, the attacker supplies a malicious τ_A using branch A that collides with the victim's branch V causing V to speculate with τ_A. With the SpectreRSB, the attacker places a malicious return address τ_A on the stack that the victim speculates with. In both cases, the target the victim will use from the STBTB or STRSB is now $\tau_V = \varphi_a \oplus \tau_A \oplus \varphi_v$. If there is a Spectre gadget located in the victim's address space at address G, the attack is successful if $\tau_V = G$. Due to A3, the attacker does not have knowledge or control of φ_a or φ_v; consequently, the only variable the attacker can change is the address of τ_A to make $\tau_V = G$. The probability that τ_A results in $\tau_V = G$ is $\frac{1}{\Omega_{STBTB}}$ or $\frac{1}{\Omega_{STRSB}}$. As such, the attacker must execute $\frac{\Omega_{STBTB}}{2}$ or $\frac{\Omega_{STRSB}}{2}$ different τ_A values to have a 50% chance of successfully executing their target injection attack. Each incorrect τ_A will result in the misprediction counter decrementing towards zero.

4.5.1.2 Reuse-based Attacks

Address mappings are randomized so that there is only a probability that an arbitrary A and V will collide in the STBPU. Even though A and V are mapped with $R_{1..4,t,p}$, the probability that attacker branch A collides with victim branch V in the STBTB/STPHT is not bound by birthday attack complexity because V is a static, specific address. The probability of collision is $P(A \Rightarrow V) = (\frac{1}{I})(\frac{1}{TO})$. Note, we break up the probability that A and V are in the same set vs. the probability that A and V have matching tag and offsets because tag/offset comparisons are only done if A and V are in the same set. This adds uncertainty for reuse-based side channels where the attacker wishes to determine the direction of V since a lack of misprediction by A or V could mean that A and V do not collide, or that V was not taken. To increase the probability that an arbitrary A collides with a static V, the attacker can execute a set of branches $S_B = \{b_1, ..., b_n\}$ where n is large so that one branch in S_B might collide with V. The probability that one of the branches in S collides with V is $P(S_B \Rightarrow V) = \sum_{i=1}^{n} P(S_{B_i} \Rightarrow V)$. However, noise is added using this method because it is possible that branches in S_B will collide with each other. The probability that two branches in S_B collide can be approximated with birthday attack complexity because the branches in S_B are arbitrary.

In order to ensure that no branches in S_B collide with any other branch in S_B, the attacker execute the following steps: i) Choose a new branch b_{new} with a new address in attacker's address space. ii) For every branch b_i in S_B, execute b_i and b_{new}. iii) If no MISP. between b_i and b_{new}, $S_B = S_B \cup \{b_{new}\}$. In order to achieve a 50% probability of collision between A and a branch in S_B, the size of S_B must be $\frac{ITO}{2}$. The number of MISPs M and evictions E generated whilst generating S_B of size $n = \frac{ITO}{2}$ can be approximated as follows:

$$
\begin{aligned}
M &\approx \sum_{i=0}^{n} \sum_{j=0}^{j=i} \frac{1}{\sqrt{\frac{\pi}{2} I}} \cdot \frac{1}{\sqrt{\frac{\pi}{2} TO}} = \frac{n(n+1))}{2\sqrt{\frac{\pi}{2} I} \cdot \sqrt{\frac{\pi}{2} TO}} \\
E &\approx \frac{ITO}{2} - IW
\end{aligned}
\tag{4.2}
$$

Note the reuse-based side channel attacks on PHT do not generate evictions. The size of the STBTB is IW which is significantly smaller than $\frac{ITO}{2}$, so entries in the BTB will constantly be evicted as the attacker grows S_B.

Attacks such as BranchScope [44] and BlueThunder [62] are viable against processors using hybrid directional predictors largely due to the inclusion of a base directional predictor in these hybrid BPUs. Due to the complexity of TAGE tables and Perceptron weights, it is *significantly* easier to maliciously modify the base directional predictor than the complex TAGE/Perceptron structures. Since the remapping mechanisms used in our TAGE/Perceptron structures are different than the remapping functions used for the base directional predictor, little information is gained by an attacker observing mispredictions from both the base and complex directional components. Due to A1, an attacker will not know which TAGE bank or Perceptron weight set produced a prediction. The thresholds for re-randomization stemming from mispredictions from the directional predictor are based on the *least* complex attack on the directional predictor. More complex attacks will be affected by re-randomization to a greater extent.

4.5.1.3 Same Address Space Attacks

Recently discovered same address space attacks [164] are classified as target injection attacks, but in this case both A and V are located inside the attacker's address space. As such, encrypting the target of A with φ_a provides no security because V will decrypt τ_A with φ_a. However, due to R_i, there is only a probability that A and V will collide; this probability is the same as for reuse-based attacks. Therefore, the number of mispredictions and evictions generated while performing a same address space attack are also approximated by Equation (4.2).

4.5.1.4 Eviction-based Attacks

The attacker cannot deterministically create BTB eviction sets without knowing ψ_a since address mappings change when ψ_a is re-randomized. With W_{stbtb} ways, detecting an eviction in an arbitrary set requires $W_{stbtb} + 1$ colliding branches (same index, different tag

and/or offset). The attacker wants to fill STBTB sets so that if V is executed, it disturbs one of the attacker's primed sets. To increase the chances that V will enter a primed set, the attacker must prime as many sets as possible. Assuming the ideal case when the attackers does not have conflicts between their own branches, they need to cover $P * I$ sets to achieve P probability of a successful attack. For example, the probability that A enters the same set as a static V is $\frac{1}{I}$, so to have a 50% chance of priming the set V enters, the attacker must prime $\frac{I}{2}$ sets. Naïvely, the probability of randomly guessing W_{stbtb} branches to form a single set of branches S_e that enters the same STBTB set is:

$$P(S_e) = \frac{1}{I^{W_{stbtb}-1}} \tag{4.3}$$

Since this probability is not favorable, the attacker could apply a fast algorithm GEM [116] to construct every eviction set. The attackers uses GEM because bottom-up strategies like PPP becomes less efficient without a partitioned randomized structure [114] or specific cache conditions [24, 28, 139]. We assume the ideal scenario for the attacker is when most of the branches tested follow a perfect uniformity. In this case, given a particular branch, the probability to have W branches belonging to the same set is directly related to the total number of test entries. For instance, there is a 50% probability that in a group of $\frac{IW}{2}$ branches that at least W branches share the same index. Thus, in order to achieve P attack rate, the attacker needs to test at least PIW branches as the initial set since the total attack lines in L in GEM. (E.g., $L > 44$ for an efficient GEM in [116]). With the original setting in GEM, the attacker sets the group size $G = W + 1$ and starts to eliminate groups of branches. Although the total branch accesses will be approximately $2.3 \cdot W \cdot L$, the total eviction number will be less as the majority of the probe during each iteration will be hit. Since the probability that each group will produce an eviction is approximately equal to $1 - 1/e$. The evictions generated by testing will be negligible as $(W + 1) \cdot 1 - \frac{1}{e} \cdot n$ since the total rounds n for GEM converge on the list of conflicting lines are relatively small. However, when first placing L branches, the attacker has to trigger the same amount

of evictions. Summarizing the procedure to construct required eviction sets above, we can now approximate evictions numbers generated whilst building sets for P attack rate as follows:

$$E \approx PI \times (PIW + (W+1) \times (1 - \frac{1}{e}) \times 3) \tag{4.4}$$

4.5.1.5 Re-randomization Thresholds for Baseline Model

STBPU has the same parameters as the baseline Intel Skylake BPU. The BTB has 8 ways and 512 sets. The stored entries have a compressed 8-bit tag and a 5 bit offset. The PHT has 1 way and 2^{14} sets. Using Equation 4.2, the number of mispredictions and evictions an attacker will trigger before a successful reuse-based side channel attack on BTB is 6.9×10^8 and $\approx 2^{21}$, respectively. Correspondingly, for a PHT reuse-based side channel, the number of triggered mispredictions is $\approx 8.38 \times 10^5$. For a BTB eviction-based side channel, the average number of triggered evictions is $\frac{I}{2}$ or 5.3×10^5 per Equation 4.4. For Spectre V2 and SpectreRSB, the number of triggered mispredictions is $\approx 2^{31}$. To prevent attacks, we use the lowest misprediction and eviction thresholds as the upper bounds for re-randomization of ST when evaluating the performance of STBPU.

4.5.1.6 Denial-of-Service Attack on STBPU

While the primary goal of a typical attacker is to reveal some sensitive data via a side channel or speculative execution attack, they can also attempt to perform a denial-of-service (DoS) attack. In this attack, the goal is to cause an abnormal slowdown of a victim process by triggering excessive branch mispredictions. We consider two DoS attack scenarios: i) Eviction-based: attacker attempts to evict from BPU data associated with a branch that is critical for the victim's performance. ii) Reuse-based: attacker fills BTB with bogus data hoping to make the victim speculatively execute code at a wrong address causing a delay due to the recovery from incorrect speculative execution. On high level, STBPU makes both of these attacks more challenging because they rely on branch instruction collisions which are difficult to create in STBPU. Now we will discuss each attack in more detail.

STBPU cannot eliminate the possibility of the first attack because, in STBPU, internal data structures such as BTB remain to be shared. However, the attack becomes more difficult to carry out with STBPU. Since the victim and attacker are guaranteed to use different STs, the attacker must default to a brute force. Due to unknown branch-to-BTB mappings, finding eviction sets becomes a difficult task. Since BTB is a set-associative structure, to guarantee eviction of a certain entry, the attacker needs to find n branches mapped into the same set, where n is the number of ways in BTB. Since the attacker is blind, the attacker must rely on constantly executing a large number of branches hoping to evict the victim's entries.

The second attack is very difficult in the case of STBPU. In order to cause a hit in BTB, the attacker's and victim's branches need to have the same index, tag, and offset after they are remapped by STBPU mechanisms with different STs. Based on our analysis above, such an event is unlikely to happen. Moreover, because the stored address is encrypted with the ST of a different process, the predicted address would most likely point to an invalid address. Thus, erroneous speculative execution would not happen.

4.6 Evaluation of STBPU Design

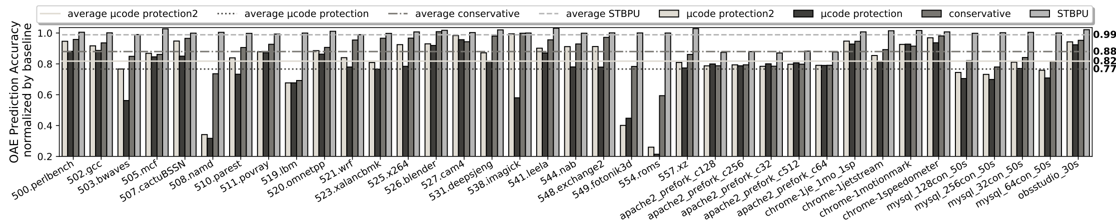


Figure 4.3: Overall branch prediction accuracy: STBPU against other secure BPU models

Evaluating BPU design under realistic conditions is a challenging task. Firstly, sharing BPU resources creates various possibilities for branch conflicts, which directly affect prediction accuracy. Moreover, microcode-based protections, such as Intel's IBRS and IBPB are triggered by system events such as mode and context switches. These mitigations can flush BPU resources to prevent BPU training or state leakage between user and kernel

processes. Workloads that involve frequent system calls and interrupts may experience performance degradation and negatively affect other programs executing on the same core. Workloads involving frequent system calls and interrupts may see performance degradation, also can potentially affect other programs executing on the same core. Contrarily, standard benchmark suites, often compute-bound and not subject to frequent system or library calls, may fall short in reflecting such impacts. Thus, an effective evaluation environment requires capturing system-wide events and incorporating real applications. A trace-based simulation is a logical choice for this. Meanwhile, understanding the complex performance side effects caused by branch mispredictions and evictions require precise performance data (e.g., IPC) using cycle -accurate simulators. To address the abovementioned aspects, we evaluate STBPU using two distinct simulation frameworks.

First, we utilize the Intel processor trace (PT) technology to collect large amounts of branch instruction traces captured from different workloads within the same CPU physical core, including user applications that cause frequent mode switches and context switches and the SPEC benchmarks. These traces then will be passed through an in-house BPU simulator with the BPU baseline found in the Intel Skylake processor. The simulator also runs different secure models such as STBPU and reports prediction accuracy. Secondly, to evaluate fine-grained microarchitectural performance effects, we implemented the STBPU mechanisms inside gem5 [23] and conducted simulations in syscall-emulation (SE) mode using DerivO3CPU model with configurations that mimic similar modern processors. The detailed configuration is listed in Table 4.3. All gem5 simulations were performed by simulating 110 million instructions with a warm-up of 10 million instructions.

ISA	Single thread: X86-64, 3.4GHz; SMT: Alpha, 3.4GHz
BPU	BTB entries: 4096, 8-way, RAS size: 16
Core	8-issue, OoO, IQ/LQ/SQ entries: 64/32/32, ROB: 192, ITLB/DTLB: 64/64
Cache	L1-I/L1-D: 32KB/32KB both 8-way, L2: 256KB 4-way, LLC: 4MB 16-way

Table 4.3: Parameters used in gem5 simulation

4.6.1 Re-randomization Threshold

In Section 4.5, we demonstrate the misprediction and eviction thresholds for ST re-randomization when various STBPU attacks have a P attack success rate. For BranchScope attack, to have a 50% chance of success, the number of triggered mispredictions is estimated at $\approx 8.38 \times 10^5$. For a BTB eviction-based side channel attack, the number of triggered evictions is $\approx 5.3 \times 10^5$. These are the lower-bound numbers of mispredictions and evictions triggered by any attack discussed in this work. We aim to re-randomize ST well before the attacker has a reasonable probability of a successful attack. To do so, we utilize results from the previously discussed security analysis and derive the re-randomization thresholds as follows. We first denote the attack complexity C as the least number of evictions or mispredictions that the attack needs to trigger to succeed with a 50% chance. Please note that we use 50% probability rather than 100% since on average the attacker will succeed with half the number of attempts needed for the fully exhaustive key search. Let the variable r be the attack difficulty factor and Γ be the re-randomization threshold. As such, $\Gamma = r \cdot C$. An attack has a 50% success rate when $r = 1$. For instance, if $r = 0.1$, then the re-randomization thresholds for mispredictions and evictions are set to 8.3×10^4 and 5.3×10^4, while 4.15×10^4 and 2.65×10^4 when $r = 0.05$. For further experiments, we set r to 0.05 and derive the re-randomization thresholds from this value as it offers strong security guarantees with a low impact on performance.

4.6.2 STBPU Performance Evaluation

4.6.2.1 Prediction Accuracy with real branch trace

We evaluate the STBPU impact on BPU accuracy and compare it to existing naïve protections modeled after microcode protections based on flushing or partitioning BPU resources. To do so, we utilize our trace-based BPU simulator based on Intel PT technology. It avoids simulating the complex state of microarchitectural components. Instead, it is designed to allow rapid testing of BPU models using branch traces from a live system

running a variety of real-world scenarios.

Each simulation instance is collected from an Intel Core i7-8550U machine that captures traces from a live physical core and includes any OS/library code executed, including naturally occurring context, mode switches, and interrupts. This allows realistically simulating complex cross-process BPU effects and assessing how BPU flushing or ST re-randomization affects performance. To evaluate single-process compute-bound scenarios, we collected 23 traces from different workloads in SPEC CPU 2017. In addition, we captured traces from user and server applications, including Apache2 workloads under different prefork settings, Google Chrome traces when running single or multiple browser workloads, MySQL server, and OBS Studio.

Introduced in Section 2.1, our baseline BPU model is based on recent reverse-engineering insights of Intel processors [42, 79, 164, 44, 93, 83]. We applied the ST mechanisms from Section 4.3.2 to the baseline BPU model as the STBPU implementation. We also created two models that mimic the baseline model with Intel's microcode-based protections, namely μcode protection 1 and 2, modeling IPBP+IBRS protection with and without STIBP. Please note that microcode-based protections cannot prevent branch collisions from occurring within the same context. To prevent such collisions, more structural BPU changes are required. In particular, instead of storing compressed and truncated addresses in BTB, the full 48-bit address must be stored. As a result, the number of entries the BTB is capable of storing must be reduced (assuming unchanged hardware budget). We refer to such a model as conservative, which fully prevents any known collision-based BPU attack by flushing or partitioning. Note that STBPU achieves the same security level via customizing BPU data representations, and has better performance.

The result from simulating the above five models is demonstrated in Figure 4.3 where we aggregate all the effective predictions into a single metric: overall accuracy effective (OAE). OAE counts a branch correctly predicted if all necessary target and direction predictions are correct; otherwise, it's counted as mispredicted. Figure 4.3 shows the overall accuracy of the various BPU models against the SPEC2017 benchmarks and user applications. STBPU

demonstrates an average 1.3% overall effective prediction accuracy penalty. For comparison, the microcode and the conservative BPU models suffer at least around 12% overall accuracy loss with multiple cases of nearly 30% reduction. With this, we conclude that based on the BPU accuracy data STBPU outperforms the heavy-handed [56, 144, 145] microcode protections that utilize flushing and partitioning.

4.6.2.2 Cycle Accurate Evaluation using gem5

Our next evaluation focuses on the comprehensive impact of STBPU on Out-of-Order (OoO) CPU in terms of cycle accurate performance, evaluating effects of STBPU on advanced branch predictors, and SMT performance. We tested three advanced BPU models: TAGE_SC_L_8KB, TAGE_SC_L_64KB [132], and PerceptronBP [68]. To demonstrate the consistency of accuracy between gem5 and our previous evaluation, we also ported and tested our baseline model from Section 4.6.2.1. We refer to it as SKLCond. We compared the direction prediction accuracy between SKLCond in gem5 with our previous baseline model using the same workloads. We observed on average less than 5% direction prediction difference which validates our simulator consistency.

We treated the aforementioned four BPUs as baseline models and implemented four STBPU models. In single process evaluations, we simulated each pair of STBPU models and their non-ST counterparts across 18 SPEC2017 workloads. Figure 4.4 illustrates the reduction of direction / target predictions rate and the normalized IPC between STBPU designs and their non-secure counterparts. We observe all 4 STBPU designs can achieve less than 2% reduction on average target prediction rate and less than 1.3% reduction on average of direction prediction rate. The less than 4% average IPC reduction demonstrates the high effectiveness of STBPU designs.

We used the same eight BPU models in our gem5 SMT simulations. Instead of running a single workload at a time, we grouped the individual workloads in pairs and simulated these pairs in SMT mode. In order to accurately evaluate the STBPU impacts on overall throughput, we calculated the Harmonic means (Hmeans) [102] of IPCs since each workload

is equally valued. Figure 4.5 displays the overall IPC and the impact on accuracy. We observed the ST_SKLcond models suffer the most in SMT mode. This is because running tasks in SMT mode introduces more frequent ST re-randomizations. However, the reduction of throughput is less than 5%. We believe this is because the ST_SKLcond model does not have a separate threshold register as TAGE models do for TAGE-table mispredictions. This causes more frequent direction mispredictions as shown in the first chart of Figure 4.5. This effect further affects the overall performance. On the other hand, the advanced BPU models overall retain their efficiencies with minimized accuracy reduction and throughput slowdown.

4.6.2.3 Aggressive ST Re-randomization and Performance

It is common to see a constant arms race between protection mechanisms and more advanced attack algorithms [114, 116] and new hardware vulnerabilities [164, 123], potentially improving attack rates by orders of magnitude. STBPU can withstand faster attack algorithms by tightening the ST re-randomization thresholds. While enhancing the BPU resilience agaisnt attacks, the more aggressive ST rerandomization frequencies can also affect the prediction accuracies. To study such performance impact, we experimented with lowering the r parameter. This is equivalent to assuming any future threats increase the attack efficiency to 10 times, 100 times, and even more.

To demonstrate an extreme case, we select an advanced BPU model that is sensitive to branch history loss and thus re-randomizations. We test it under SMT scenarios that are more prone to branch mispredictions and evictions. Figure 4.6 demonstrates how reducing the r parameter affects the performance of the TAGE_SC_L BPU protected with STBPU. It shows that the thresholds can be safely reduced and maintain accuracy above 95%. However, setting the threshold too low results in ST re-randomizations happening after every few hundreds of mispredictions or evictions. This practically ceases any BPU training.

We argue that the C2 constraint (uniformity) of remapping prevents STBPU from interfering with the nature of BPUs and applications having high prediction rates. On

the other hand, without uncovering both attacker and victim's STs, an attacker tends to pollute his own program with deliberately generated events like evictions, resulting in more frequent ST updates.

4.7 Related Work

Branch collision-based attacks can be mitigated in different ways, resulting into different scope of protection. For instance, to avoid leaking secret key bits from RSA secret key operation, one can rewrite the exponentiation calculation code to avoid secret-key dependent branches [14]. On the other hand, when flushing branch predictor on context switch attacks such as Spectre v2, Jump-over-ASLR and other attacks can be fully mitigated. To this end we chose to compare STBPU to the existing mitigations, other hardware-based defenses and BPU designs that can deliver similar protection scope i.e., being capable of suppressing both side channel and speculative execution attacks.

4.7.1 Existing Microcode-based Mitigations

CPU manufacturers such as Intel provide microcode update to protect exising CPUs agaisnt speculative execution attacks. These Technologies include Indirect Branch Restricted Speculation (IBRS), Indirect Branch Prediction Barrier (IBPB), and Single Threaded Indirect Branch Prediction (STIBP) [64]. IBRS prevents higher level processes from speculating with entries provided by lower level processes by flushing. IBPB provides protection by flushing the contents of the BPU on context switches. While effectively removing the interference from the branch prediction on other processes, flushing the BPU removes useful history often come with a very costly performance overhead [137, 113]. As a result, in practice, they are not used to their full extent and are only enabled by the OS in critical cases such as protecting kernel or a handful of processes. Additionally, recent research demonstrated exploitable branch collisions within same address space [164, 123]. Therefore enforcing security only during context switch is not complete. STIBP logically segments the BPU so that branch prediction of on the same physical assemblage does not

interfere with each other. In contrast, STBPU works regardless of context switch activity. We choose to compare performance of STBPU to protection mechanisms derived from such protections as they, when properly activated, can provide strong BPU isolation.

4.7.2 Other Defense Directions against Spectre Attacks

While many defenses have been proposed from different directions, they also all suffer from various limitation such as stopping too few Spectre variant or BPU attacks or drawing too much performance overheads.

CEASER [115], SCATTERCACHE [156], and RPCache [153] use hardware supported block ciphers to change addresses mapping in cache; STBPU remappings deviate from these works with better protection range and fits in more restricted hardware timing constraints. SafeSpec [71] introduces multiple shadow structures to hide the effects of erroneous speculative execution. The effects of speculation are only visible once speculation has been confirmed to be correct. InvisiSpec [160] prevents attackers from seeing changes to caches as a result of speculative execution. Similar to the other protections that focus on monitor and control data visibility in and out of cache, these cannnot stop a range of the attacks through other covert-channels [123]. Given that, STBPU does not use the cache as the defensive vector, and instead, draws its security from continually changing how addresses are mapped and how branch look-up prediction information is stored in the BPU. Perspectron [105] identifies Spectre and cache based attacks before execution through the use of perceptorns, but does not provide any follow-up protections. Conditional Speculation [88] splits up condition branches into safe and unsafe instructions only allows speculative execution of safe conditionals while STBPU allows safe acceleration on all branches. NDA [155] protects against Meltdown and Spectre attacks by preventing data from propagating to their dependencies speculatively. This approach differs from our design as it does not change the data representations or mappings in the BPU, only what happens after a branch is taken.

4.7.3 Existing Secure Designs of BPU

Several previous academic works proposed BPU modifications to protect against side channel and speculative execution attacks. BRB [149] stores and reloads the entire history of the directional predictor for each process, effectively mitigating PHT collision-based attacks such as BranchScope. BSUP [85] first encrypts the PC and then encrypts the entries of BPU, making it unsuitable for SMT processors.

Zhao et al. [168] encode branch contents (directions and destination histories) and indexes using thread-private random numbers to achieve isolation between threads or privilege levels. Their approaches re-generate random numbers upon context and mode switches, which cannot defend against the transient execution attacks from same-address-space [164, 158]. Besides, our work implements ST re-randomization based on BPU events allowing efficient branch history retention.

The BPU in the Samsung Exynos processor is also protected with XOR-based encryption as branch history data enhancement [52]. Since this mechanism aims to prevent speculative execution attacks such as Spectre variant 2, Exynos only encrypts stored branch targets of indirect branch instructions and returns. However, other forms of branch collisions may still result in side channel leakage [168]. Additionally, in Exynos, an output of the hash function serves as a key for encrypting branch target data. It is derived from a number of process and machine-specific inputs. In our work, the OS is given more flexibility for managing the ST, which allows selective branch history sharing, adjustment of re-randomization frequency, and enforcing BPU isolation for various types of software entities such as sandboxes and libraries.

4.8 Conclusions & Accessibility

We presented the STBPU, a safe branch prediction design that defends against BPU side channel and speculative execution attacks. We performed an systematization of BPU related attacks and provided a detailed security analysis agaisnt the most recent attack

models. While retaining secuity, we demonstrate the high effeciency across both real-world models and advanced models. We plan to release our simulation tools, gem5 modification, and more details in STBPU desgin, evaluations, and hw estimation.

We then evaluated the STBPU against the baseline Intel Skylake BPU with/without current microcode protections, and against a theoretical conservative BPU design that has similar security benefits to STBPU. Finally, we added out STBPU principles into modern branch predictor designs. Out 8KB and 64KB ST-TAGE-L-SC models achieved high performance with less than 2% IPC overhead, and our 64KB ST-Perceptron model received less that 11% overhead. This work has been summarized and published in DSN 2022 [165].

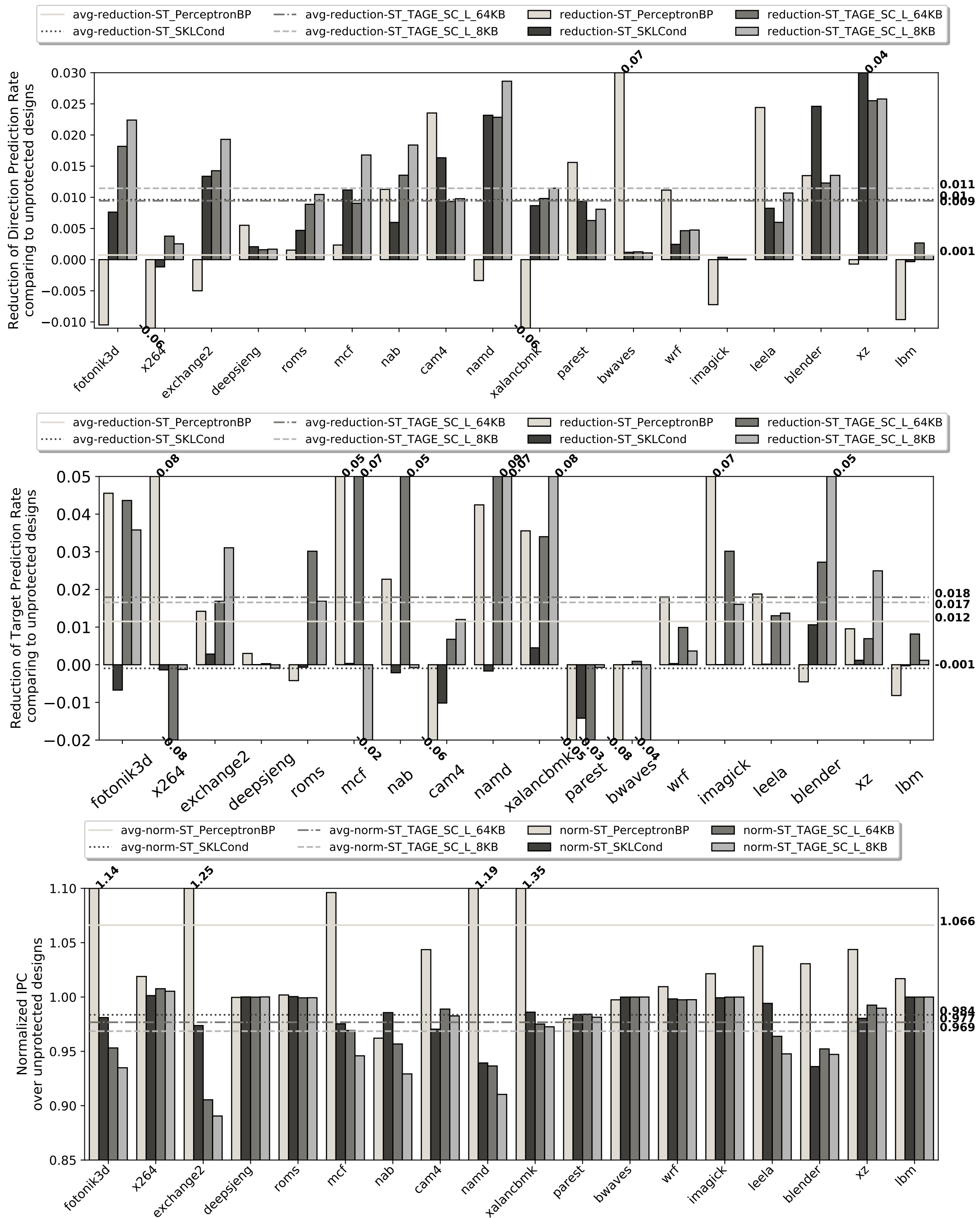


Figure 4.4: STBPU single workload evaluation in gem5

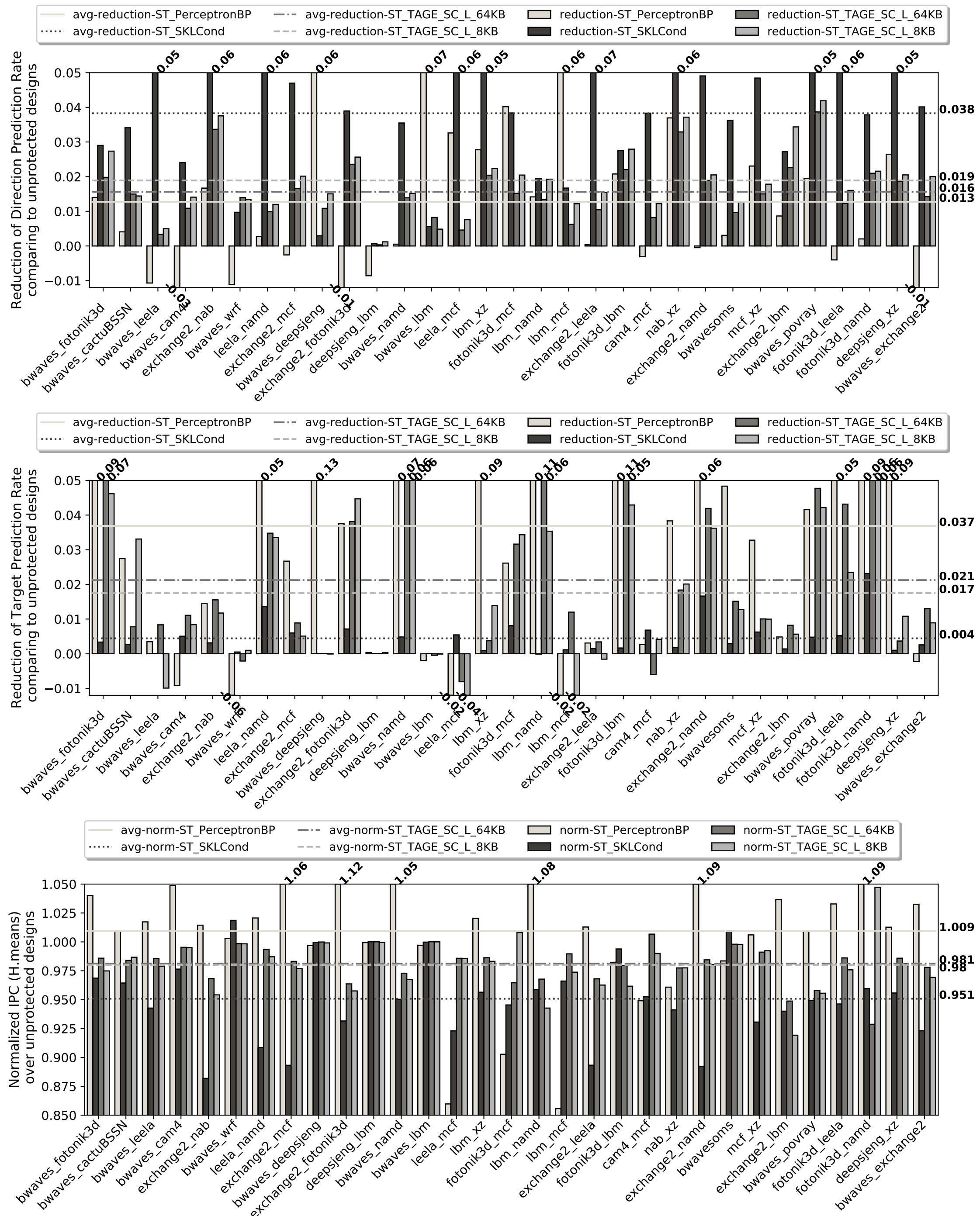


Figure 4.5: Gem5 Multi-workload (SMT) Evaluation of STBPU

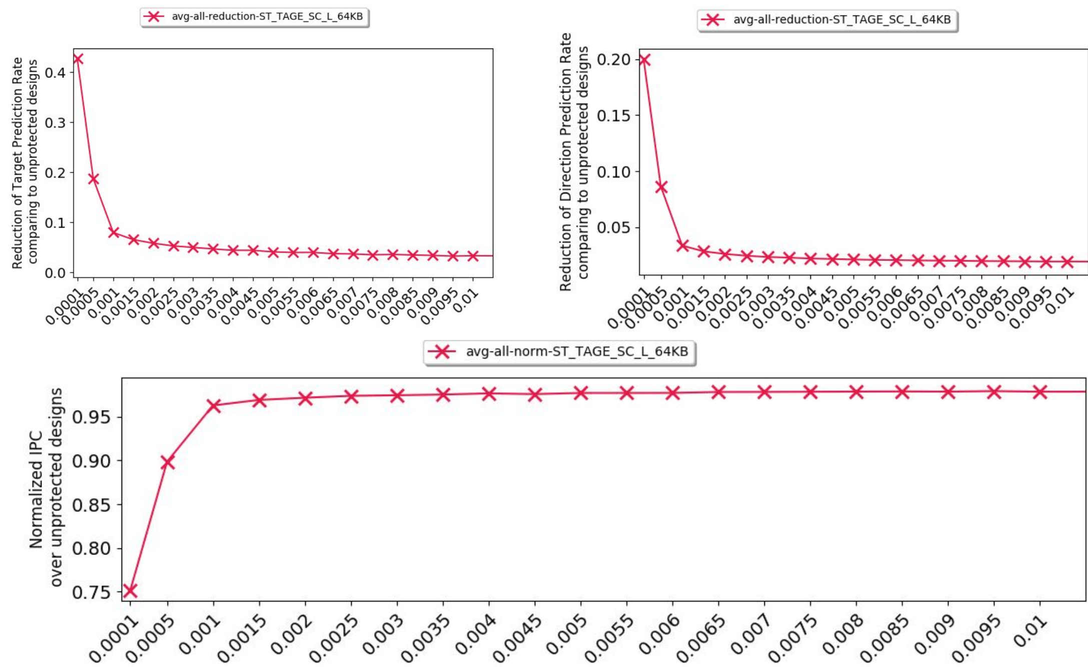


Figure 4.6: Effects on performance when using more aggressive re-randomization thresholds with the TAGE_SC_L_64KB BPU, result are averaged from 42 combinations of SPEC CPU 2017 workload pairs. The X-axis represents the r parameter.

Chapter 5

Conclusion

This dissertation has presented a comprehensive exploration of transient execution vulnerabilities, side-channel attacks, and defenses, with a particular focus on the role of branch prediction units (BPUs) in modern microprocessors. Through a detailed examination of the mechanisms behind branch prediction and speculative execution, we have uncovered new vulnerabilities and proposed innovative defense mechanisms. Our work is situated within the broader context of ongoing efforts to secure microprocessors against a range of speculative execution attacks and side-channel attacks, which have highlighted the security challenges posed by modern microarchitectural optimization techniques.

5.1 Main Contributions

Our research contributions can be summarized as follows:

1. **Exploring Branch Predictors for Constructing Transient Execution Trojans**: We have identified previously unknown vulnerabilities within BPUs that can be exploited to construct transient execution trojans. By reverse-engineering branch predictors in recent x86_64 processors, we have unveiled hidden branch prediction mechanisms that allow for the creation of these trojans, demonstrating their stealthiness and practicality.

2. **STBPU: A Reasonably Secure Branch Predictor Unit**: We proposed the Secret Token Branch Predictor Unit (STBPU), a novel BPU design that offers robust defense against collision-based speculative execution attacks and BPU side channels. STBPU incorporates unique branch representation for each software entity, ensuring strong isolation and significantly reducing the potential for speculative execution attacks without imposing substantial performance overheads. Through security analysis and performance evaluation, we have demonstrated the effectiveness of STBPU in defending against a wide range of attacks while maintaining minimal performance impact.

5.2 Implications and Future Directions

The findings from this dissertation have implications for the design and security of future microprocessors. By highlighting the vulnerabilities within BPUs and proposing effective defenses, we contribute to the ongoing efforts to balance performance optimizations with security requirements. The STBPU design, in particular, represents a direction for developing secure microarchitectures that do not compromise on performance.

For future research, several directions are worth exploring:

- **Further Refinement of STBPU**: While STBPU represents a solution in BPU security, further research could further explore optimizations to enhance its performance and security. Additionally, the adaptability of STBPU to other speculative execution vulnerabilities beyond those addressed in this dissertation requires further investigation.
- **Broadening the Scope of Secure Microarchitecture Designs**: Beyond BPUs, other components of the CPU microarchitecture may also harbor vulnerabilities that could be exploited through speculative execution attacks. Future work should aim to identify these vulnerabilities and develop comprehensive defense mechanisms.
- **Cross-layer Security Approaches**: As demonstrated by the complexity of speculative execution attacks, securing modern microprocessors requires cross-layer ap-

proaches and in-depth defenses that spans hardware design, system software, and application software. In comparison to proposing future hardware design, securing existing and legacy devices is also critical and challenging. Integrating the insights from this dissertation with efforts at other layers of the computing stack presents a promising research direction.

In conclusion, this dissertation contributes to the critical task of securing microprocessors against speculative execution attacks and CPU side-channel attacks. By focusing on the vulnerabilities associated with branch prediction and speculative execution, we have shed light on the challenges and opportunities for designing secure microarchitectures. The proposed STBPU design represents a direction in the quest for secure, high-performance computing. As the computing landscape continues to evolve, it is imperative that security considerations remain at the forefront of microprocessor design and optimization strategies.

Bibliography

[1] Intel® 64 and IA32 architectures performance monitoring events, 2017. https://software.intel.com/sites/default/files/managed/8b/6e/335279_performance_monitoring_events_guide.pdf.

[2] Amd. software techniques for managing speculation on amd processors., 2018.

[3] Detecting Spectre vulnerability exploits with static analysis, march 2018.

[4] Intel analysis of speculative execution side channels, 2018. https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf.

[5] LWN.net: Finding Spectre vulnerabilities with smatch, April 2018. https://lwn.net/Articles/752408/.

[6] Intel® 64 and IA-32 architectures optimization reference manual, 2019. https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf.

[7] Wikichip:skylake(client)-microarchitectures-intel, 2019. https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client).

[8] Nael Abu-Ghazaleh, Dmitry Ponomarev, and Dmitry Evtyushkin. How the spectre and meltdown hacks really worked. *IEEE Spectrum*, 56(3):42–49, 2019.

[9] Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 110–124. Springer, 2010.

[10] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security*, pages 312–320. ACM, 2007.

[11] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *Cryptographers' Track at the RSA Conference*, pages 225–242. Springer, 2007.

[12] Shaizeen Aga and Satish Narayanasamy. Invisimem: Smart memory defenses for memory bus side channel. In *ACM SIGARCH Computer Architecture News*, volume 45, pages 94–106. ACM, 2017.

[13] Murugappan Alagappan, Jeyavijayan Rajendran, Miloš Doroslovački, and Guru Venkataramani. Dfs covert channels on multi-core platforms. In *2017 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 1–6. IEEE, 2017.

[14] Monjur Alam, Haider Adnan Khan, Moumita Dey, Nishith Sinha, Robert Callan, Alenka Zajic, and Milos Prvulovic. One&done: A single-decryption em-based attack on openssl's constant-time blinded {RSA}. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 585–602, 2018.

[15] Eran Altshuler, Oded Lempel, Robert Valentine, and Nicolas Kacevas. Preventing a read of a next sequential chunk in branch prediction of a subject chunk, February 6 2007. US Patent 7,174,444.

[16] Nadav Amit, Fred Jacobs, and Michael Wei. Jumpswitches: Restoring the

performance of indirect branches in the era of spectre. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 285–300, 2019.

[17] DW Anderson, FJ Sparacio, and Robert M Tomasulo. The ibm system/360 model 91: Machine philosophy and instruction-handling. *IBM Journal of Research and Development*, 11(1):8–24, 1967.

[18] Marco Angelini, Graziano Blasilli, Pietro Borrello, Emilio Coppa, Daniele Cono D'Elia, Serena Ferracci, Simone Lenti, and Giuseppe Santucci. Ropmate: Visually assisting the creation of rop-based exploits. In *2018 IEEE Symposium on Visualization for Cyber Security (VizSec'18)*, 2018.

[19] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):50, 2018.

[20] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks. In *USENIX Security*, 2022.

[21] Daniel J Bernstein. Cache-timing attacks on aes. 2005.

[22] Sarani Bhattacharya, Clémentine Maurice, Shivam Bhasin, and Debdeep Mukhopadhyay. Branch prediction attack on blinded scalar multiplication. *IEEE Transactions on Computers*, 69(5):633–648, 2019.

[23] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib Bin Altaf, Nilay Vaish, Mark Hill, and David Wood. The gem5 simulator. *SIGARCH Computer Architecture News*, 39:1–7, 08 2011.

[24] Rahul Bodduna, Vinod Ganesan, Patanjali Slpsk, Kamakoti Veezhinathan, and Chester Rebeiro. Brutus: Refuting the security claims of the cache timing randomization countermeasure proposed in ceaser. *IEEE Computer Architecture Letters*, 19(1):9–12, 2020.

[25] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe. Present: An ultra-lightweight block cipher. In *Cryptographic Hardware and Embedded Systems - CHES 2007*, Pascal Paillier and Ingrid Verbauwhede, editors, pages 450–466, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[26] Andrey Bogdanov, Miroslav Knežević, Gregor Leander, Deniz Toz, Kerem Varici, and Ingrid Verbauwhede. spongent: A lightweight hash function. In *Cryptographic Hardware and Embedded Systems – CHES 2011*, Bart Preneel and Tsuyoshi Takagi, editors, pages 312–325, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[27] Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knezevic, Lars R Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, et al. Prince–a low-latency block cipher for pervasive computing applications. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 208–225. Springer, 2012.

[28] Thomas Bourgeat, Jules Drean, Yuheng Yang, Lillian Tsai, Joel Emer, and Mengjia Yan. Casa: End-to-end quantitative security analysis of randomly mapped caches. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1110–1123. IEEE, 2020.

[29] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, and

Srinivas Devadas. Mi6: Secure enclaves in a speculative out-of-order processor. pages 42–56, 2019.

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

[31] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. pages 249–266, 2019.

[32] Po-Yung Chang, Eric Hao, Tse-Yu Yeh, and Yale Patt. Branch classification: a new mechanism for improving branch predictor performance. *International Journal of Parallel Programming*, 24(2):133–158, 1996.

[33] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. pages 142–157, June 2019.

[34] Jie Chen and Guru Venkataramani. Cc-hunter: Uncovering covert timing channels on shared processor hardware. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 216–228. IEEE, 2014.

[35] Sanchuan Chen, Fangfei Liu, Zeyu Mi, Yinqian Zhang, Ruby B Lee, Haibo Chen, and XiaoFeng Wang. Leveraging hardware transactional memory for cache side-channel defenses. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 601–608. ACM, 2018.

[36] James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic

taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 196–206. ACM, 2007.

[37] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 45–60. IEEE, 2009.

[38] Jonas Depoix and Philipp Altmeyer. Detecting spectre attacks by identifying cache side-channel attacks using machine learning. *Advanced Microkernel Operating Systems*, page 75, 2018.

[39] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. Cacheaudit: A tool for the static analysis of cache side channels. *ACM Transactions on Information and System Security (TISSEC)*, 18(1):4, 2015.

[40] Marius Evers, Po-Yung Chang, and Yale N Patt. Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches. In *ACM SIGARCH Computer Architecture News*, volume 24, pages 3–11. ACM, 1996.

[41] Dmitry Evtyushkin and Dmitry Ponomarev. Covert channels through random number generator: Mechanisms, capacity estimation and mitigations. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 843–857. ACM, 2016.

[42] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over aslr: Attacking branch predictors to bypass aslr. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, pages 40:1–40:13, Piscataway, NJ, USA, 2016. IEEE Press.

[43] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Under-

standing and mitigating covert channels through branch predictors. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(1):10, 2016.

[44] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, Dmitry Ponomarev, et al. Branchscope: A new side-channel attack on directional branch predictor. In *ACM SIGPLAN Notices*, volume 53, pages 693–707. ACM, 2018.

[45] Andrew Ferraiuolo, Mark Zhao, Andrew C Myers, and G Edward Suh. Hyperflow: A processor architecture for nonmalleable, timing-safe information flow security. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1583–1600. ACM, 2018.

[46] Michael J Flynn. Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9):948–960, 1972.

[47] Gordon Fraser, Franz Wotawa, and Paul E Ammann. Testing with model checkers: a survey. *Software Testing, Verification and Reliability*, 19(3):215–261, 2009.

[48] GCC. 6.39 attribute syntax. `https://gcc.gnu.org/onlinedocs/gcc/Attribute-Syntax.html`, 2020.

[49] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 8(1):1–27, 2018.

[50] Qian Ge, Yuval Yarom, Frank Li, and Gernot Heiser. Your processor leaks information-and there's nothing you can do about it. *arXiv preprint arXiv:1612.04474*, 2016.

[51] Abraham Gonzalez, Ben Korpan, Ed Younis, and Jerry Zhao. Spectrum: Classifying, replicating and mitigating spectre attacks on a speculating risc-v microarchitecture. 2018.

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

[53] Michael Gschwind. Polymorphic branch predictor and method with selectable mode of prediction, April 21 2009. US Patent 7,523,298.

[54] Marco Guarnieri, Boris Köpf, José F Morales, Jan Reineke, and Andrés Sánchez. Spectector: Principled detection of speculative information flows. *arXiv preprint arXiv:1812.08639*, 2018.

[55] Marco Guarnieri, Boris Köpf, José F Morales, Jan Reineke, and Andrés Sánchez. Spectector: Principled detection of speculative information flows. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2020.

[56] Red Hat. Controlling the performance impact of microcode and security patches for cve-2017-5754 cve-2017-5715 and cve-2017-5753 using red hat enterprise linux tunables. *Red Hat Customer Portal*. https://access.redhat.com/articles/3311301/.

[57] Andrei Homescu, Michael Stewart, Per Larsen, Stefan Brunthaler, and Michael Franz. Microgadgets: size does matter in turing-complete return-oriented programming. In *Proceedings of the 6th USENIX conference on Offensive Technologies*, pages 7–7. USENIX Association, 2012.

[58] Jann Horn. Reading privileged memory with a side-channel. *Project Zero*, January 2018. `https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html`.

[59] Bradley D Hoyt, Glenn J Hinton, David B Papworth, Ashwani K Gupta, Michael A Fetterman, Subramanian Natarajan, Sunil Shenoy, and

Reynold V D'sa. Method and apparatus for implementing a set-associative branch target buffer, November 12 1996. US Patent 5,574,871.

[60] N. Hua, E. Norige, S. Kumar, and B. Lynch. Non-crypto hardware hash functions for high performance networking asics. In *2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems*, pages 156–166, Oct 2011.

[61] Casen Hunger, Mikhail Kazdagli, Ankit Rawat, Alex Dimakis, Sriram Vishwanath, and Mohit Tiwari. Understanding contention-based channels and using them for defense. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 639–650. IEEE, 2015.

[62] Tianlin Huo, Xiaoni Meng, Wenhao Wang, Chunliang Hao, Pei Zhao, Jian Zhai, and Mingshu Li. Bluethunder: A 2-level directional predictor based side-channel attack against sgx. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(1):321–347, Nov. 2019.

[63] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache attacks enable bulk key recovery on the cloud. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 368–388. Springer, 2016.

[64] Intel. Intel analysis of speculative execution side channels, January 2018.

[65] Intel. Retpoline: A branch target injection mitigation. 2018. reference no. 337131-003.

[66] Intel. Speculative execution side channel mitigations. 2018. reference no. 336996-003.

[67] R Intel. Intel 64 and ia-32 architectures optimization reference manual. *Intel Corporation, May*, 2020.

[68] Daniel A. Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, HPCA '01, page 197, USA, 2001. IEEE Computer Society.

[69] David Kanter. Intel's sandy bridge microarchitecture. 2010.

[70] Khaled N Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Safespec: Banishing the spectre of a meltdown with leakage-free speculation. pages 1–6, 2019.

[71] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael B. Abu-Ghazaleh. Safespec: Banishing the spectre of a meltdown with leakage-free speculation. *CoRR*, abs/1806.05179, 2018.

[72] S Karen Khatamifard, Longfei Wang, Amitabh Das, Selcuk Kose, and Ulya R Karpuzcu. Powert channels: A novel class of covert communication exploiting power management vulnerabilities. In *Proceedings of the 25th IEEE International Symposium on High-Performance Computer Architecture*, 2019.

[73] Jonathan Khazam. Method and apparatus for performing power management by suppressing the speculative execution of instructions within a pipelined microprocessor, August 28 2001. US Patent 6,282,663.

[74] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. Stealthmem: System-level protection against cache-based side channel attacks in the cloud. In *USENIX Security symposium*, pages 189–204, 2012.

[75] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. Dawg: A defense against cache timing attacks in speculative execution processors. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 974–987. IEEE, 2018.

[76] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757*, 2018.

[77] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses. *CoRR*, abs/1807.03757, 2018.

[78] Paul Kocher. Spectre mitigations in microsoft's c/c++ compiler. *Retrieved August*, 3:2018, 2018.

[79] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[80] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Annual International Cryptology Conference*, pages 388–397. Springer, 1999.

[81] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. *arXiv preprint arXiv:1807.07940*, 2018.

[82] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, Baltimore, MD, 2018. USENIX Association.

[83] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael B. Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. *CoRR*, abs/1807.07940, 2018.

[84] G. Leander and A. Poschmann. On the classification of 4 bit s-boxes. In

Proceedings of the 1st International Workshop on Arithmetic of Finite Fields, WAIFI '07, pages 159–176, Berlin, Heidelberg, 2007. Springer-Verlag.

[85] Jaekyu Lee, Yasuo Ishii, and Dam Sunwoo. Securing branch predictors with two-level encryption. 17(3), August 2020.

[86] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside sgx enclaves with branch shadowing. In *26th USENIX Security Symposium, USENIX Security*, pages 16–18, 2017.

[87] Tamara Silbergleit Lehman, Andrew D Hilton, and Benjamin C Lee. Poisonivy: Safe speculation for secure memory. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 38. IEEE Press, 2016.

[88] 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)*, pages 264–276, 2019.

[89] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[90] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, pages 406–418. IEEE, 2016.

[91] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee.

Last-level cache side-channel attacks are practical. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 605–622. IEEE, 2015.

[92] John M. Intel® digital random number generator (drng) software implementation guide, Oct 2019.

[93] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. *CoRR*, abs/1807.10364, 2018.

[94] Giorgi Maisuradze and Christian Rossow. Ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 2109–2122, New York, NY, USA, 2018. ACM.

[95] Giorgi Maisuradze and Christian Rossow. Speculose: Analyzing the security implications of speculative execution in cpus. *arXiv preprint arXiv:1801.04084*, 2018.

[96] Andrea Mambretti, Matthias Neugschwandtner, Alessandro Sorniotti, Engin Kirda, William Robertson, and Anil Kurmus. Let's not speculate: Discovering and analyzing speculative execution attacks. *IBM Technical Report*, 2018.

[97] Stefan Mangard. A simple power-analysis (spa) attack on implementations of the aes key expansion. In *International Conference on Information Security and Cryptology*, pages 343–358. Springer, 2002.

[98] Yonghua Mao, Huiyang Zhou, Xiaolin Gui, and Junjie Shen. Exploring convolution neural network for branch prediction. *IEEE Access*, 8:152008–152016, 2020.

[99] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: cross-cores cache covert channel. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 46–64. Springer, 2015.

[100] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 2018.

[101] Thomas S Messerges, Ezzat A Dabbish, and Robert H Sloan. Examining smart-card security under the threat of power analysis attacks. *IEEE transactions on computers*, 51(5):541–552, 2002.

[102] Pierre Michaud. Demystifying multicore throughput metrics. *IEEE Computer Architecture Letters*, 12(2):63–66, 2012.

[103] Pierre Michaud, André Seznec, and Richard Uhlig. Trading conflict and capacity aliasing in conditional branch predictors. In *ACM SIGARCH Computer Architecture News*, volume 25, pages 292–303. ACM, 1997.

[104] Alyssa Milburn, Ke Sun, and Henrique Kawakami. You Cannot Always Win the Race: Analyzing the LFENCE/JMP Mitigation for Branch Target Injection. *arXiv preprint arXiv:2203.04277*, 2022.

[105] Samira Mirbagher-Ajorpaz, Gilles Pokam, Esmaeil Mohammadian-Koruyeh, Elba Garza, Nael Abu-Ghazaleh, and Daniel A Jiménez. Perspectron: Detecting invariant footprints of microarchitectural attacks with perceptron. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1124–1137. IEEE, 2020.

[106] Hoda Naghibijouybari and Nael Abu-Ghazaleh. Covert channels on gpgpus. *Computer Architecture Letters*, 2016.

[107] Siddika Berna Ors, Frank Gurkaynak, Elisabeth Oswald, and Bart Preneel. Power-analysis attack on an asic aes implementation. In *Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004. International Conference on*, volume 2, pages 546–552. IEEE, 2004.

[108] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers' Track at the RSA Conference*, pages 1–20. Springer, 2006.

[109] pakt. A turing complete rop compiler. `https://github.com/pakt/ropc`, 2013.

[110] Colin Percival. Cache missing for fun and profit, 2005.

[111] Chris H Perleberg and Alan Jay Smith. Branch target buffer design and optimization. *IEEE transactions on computers*, 42(4):396–412, 1993.

[112] Andrew Prout, William Arcand, David Bestor, Bill Bergeron, Chansup Byun, Vijay Gadepally, Michael Houle, Matthew Hubbell, Michael Jones, Anna Klein, et al. Measuring the impact of spectre and meltdown. pages 1–5, 2018.

[113] Andrew Prout, William Arcand, David Bestor, Bill Bergeron, Chansup Byun, Vijay Gadepally, Michael Houle, Matthew Hubbell, Michael Jones, Anna Klein, et al. Measuring the impact of spectre and meltdown. *arXiv preprint arXiv:1807.08703*, 2018.

[114] Antoon Purnal, Lukas Giner, Daniel Gruss, and Ingrid Verbauwhede. Systematic analysis of randomization-based protected cache architectures. In *42th IEEE Symposium on Security and Privacy*.

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

[116] Moinuddin K Qureshi. New attacks and defense for encrypted-address cache. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 360–371. IEEE, 2019.

[117] Martin Raab and Angelika Steger. "balls into bins" - a simple and tight analysis. In *Proceedings of the Second International Workshop on Randomization and Approximation Techniques in Computer Science*, RANDOM '98, pages 159–170, Berlin, Heidelberg, 1998. Springer-Verlag.

[118] Himanshu Raj, Ripal Nathuji, Abhishek Singh, and Paul England. Resource management for isolation enhanced cloud services. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 77–84. ACM, 2009.

[119] Matt Ramsay, Chris Feucht, and Mikko H Lipasti. Exploring efficient smt branch predictor design. In *Workshop on Complexity-Effective Design, in conjunction with ISCA*, volume 26, 2003.

[120] Lihu Rappoport, Chen Koren, Franck Sala, Ilhyun Kim, Lior Libis, Ron Gabor, and Oded Lempel. Method and apparatus for pipeline inclusion and instruction restarts in a micro-op cache of a processor, April 30 2013. US Patent 8,433,850.

[121] Lihu Rappoport, Bob Valentine, Stephan Jourdan, Yoav Almog, Franck Sala, Amir Leibovitz, Ido Ouziel, and Ron Gabor. Efficient method and apparatus for employing a micro-op cache in a processor, January 24 2012. US Patent 8,103,831.

[122] Redhat. Controlling the performance impact of microcode and security patches for cve-2017-5754 cve-2017-5715 and cve-2017-5753 using red hat enterprise linux tunables. `https://access.redhat.com/articles/3311301`, 2018.

[123] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M Tullsen, and Ashish Venkat. I see dead μops: Leaking secrets via intel/amd micro-op caches.

[124] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean,

and Magnus Själander. Efficient invisible speculative execution through selective delay and value prediction. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 723–735. IEEE, 2019.

[125] Elham Salimi and Narges Arastouie. Backdoor detection system using artificial neural network and genetic algorithm. In *2011 International Conference on Computational and Information Sciences*, pages 817–820. IEEE, 2011.

[126] Brendan Saltaformaggio, Dongyan Xu, and Xiangyu Zhang. Busmonitor: A hypervisor-based solution for memory bus covert channels. *Proceedings of EuroSec*, 2013.

[127] Felix Schuster and Thorsten Holz. Towards reducing the attack surface of software backdoors. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 851–862. ACM, 2013.

[128] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. Netspectre: Read arbitrary memory over network. pages 279–299, 2019.

[129] A. Seznec and P. Michaud. A case for (partially) tagged geometric history length branch prediction. *Journal of Instruction-level Parallelism - JILP*, 8, 01 2006.

[130] André Seznec. The l-tage branch predictor. *J. Instruction-Level Parallelism*, 9, 2007.

[131] André Seznec. A new case for the tage branch predictor. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, page 117–127, New York, NY, USA, 2011. Association for Computing Machinery.

[132] André Seznec. Tage-sc-l branch predictors again. 2016.

[133] André Seznec. Tage-sc-l branch predictors. 06 2014.

[134] Hovav Shacham et al. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *ACM conference on Computer and communications security*, pages 552–561. New York,, 2007.

[135] D Shapiro. A comparison of various methods for detecting and utilizing parallelism in a single instruction stream. In *Proceedings of the 1977 International Conference on Parallel Processing*, pages 67–76, 1977.

[136] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS*, 2015.

[137] Nikolay A Simakov, Martins D Innus, Matthew D Jones, Joseph P White, Steven M Gallo, Robert L DeLeon, and Thomas R Furlani. Effect of meltdown and spectre patches on the performance of hpc applications. *arXiv preprint arXiv:1801.04329*, 2018.

[138] Baruch Solomon, Ronny Ronen, and Doron Orenstien. Power reduction for processor front-end by caching decoded instructions, September 27 2005. US Patent 6,950,903.

[139] Wei Song, Boya Li, Zihan Xue, Zhenzhen Li, Wenhao Wang, and Peng Liu. Randomized last-level caches are still vulnerable to cache side-channel attacks! But we can fix it. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 955–969. IEEE, 2021.

[140] Mohammadkazem Taram, Ashish Venkat, and Dean 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*, pages 395–410, 2019.

[141] David Tarjan and Kevin Skadron. Merging path and gshare indexing in

perceptron branch prediction. *ACM Trans. Archit. Code Optim.*, 2(3):280–300, September 2005.

[142] Sam L Thomas, Flavio D Garcia, and Tom Chothia. Humidify: a tool for hidden functionality detection in firmware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–300. Springer, 2017.

[143] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. Meltdownprime and spectreprime: Automatically-synthesized attacks exploiting invalidation-based coherence protocols. *arXiv preprint arXiv:1802.03802*, 2018.

[144] Liam Tung. Linus torvalds: After big linux performance hit, spectre v2 patch needs curbs. *ZDNet*. https://www.zdnet.com/article/linus-torvalds-after-big-linux-performance-hit-spectre-v2-patch-needs-curbs/.

[145] Liam Tung. Windows 10 will banish spectre slowdowns with google's retpoline patch. *ZDNet*. https://www.zdnet.com/article/windows-10-will-banish-spectre-slowdowns-with-googles-retpoline-patch/.

[146] Paul Turner and Google Project Zero. Retpoline: a software construct for preventing branch-target-injection. *Google Help*, 2018.

[147] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael M Swift. Scheduler-based defenses against cross-vm side-channels. In *USENIX Security Symposium*, pages 687–702, 2014.

[148] Pepe Vila, Boris Köpf, and José F Morales. Theory and practice of finding eviction sets. pages 39–54, 2019.

[149] Ilias Vougioukas, Nikos Nikoleris, Andreas Sandberg, Stephan Diestelhorst, Bashir M Al-Hashimi, and Geoff V Merrett. Brb: Mitigating branch

predictor side-channels. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 466–477. IEEE, 2019.

[150] Jack Wampler, Ian Martiny, and Eric Wustrow. Exspectre: Hiding malware in speculative execution. In *26th Annual Network and Distributed System Security Symposium*, San Diego, CA, 2019. NDSS-Symposium.

[151] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead defense against spectre attacks via program analysis. *IEEE Transactions on Software Engineering*, 2019.

[152] Zhenghong Wang and Ruby B Lee. New cache designs for thwarting software cache-based side channel attacks. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 494–505. ACM, 2007.

[153] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 494–505, New York, NY, USA, 2007. ACM.

[154] Zhenghong Wang and Ruby B Lee. A novel cache architecture with enhanced performance and security. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pages 83–93. IEEE Computer Society, 2008.

[155] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F. Wenisch, and Baris Kasikci. Nda: Preventing speculative execution attacks at their source. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 572–586, New York, NY, USA, 2019. Association for Computing Machinery.

[156] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. Scattercache: Thwarting cache attacks via

cache set randomization. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 675–692, Santa Clara, CA, August 2019. USENIX Association.

[157] Chris Wysopal, Chris Eng, and Tyler Shields. Static detection of application backdoors. *Datenschutz und Datensicherheit-DuD*, 34(3):149–155, 2010.

[158] Wenjie Xiong and Jakub Szefer. Survey of transient execution attacks and their mitigations. *ACM Computing Surveys (CSUR)*, 54(3):1–36, 2021.

[159] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. Invisispec: Making speculative execution invisible in the cache hierarchy. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 428–441. IEEE, 2018.

[160] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. Invisispec: Making speculative execution invisible in the cache hierarchy. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, pages 428–441, Piscataway, NJ, USA, 2018. IEEE Press.

[161] Tse-Yu Yeh and Yale N Patt. Two-level adaptive training branch prediction. In *Proceedings of the 24th annual international symposium on Microarchitecture*, pages 51–61. ACM, 1991.

[162] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher 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*, MICRO '52, page 954–968, New York, NY, USA, 2019. Association for Computing Machinery.

[163] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher 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*, MICRO '52, page 954–968, New York, NY, USA, 2019. Association for Computing Machinery.

[164] Tao Zhang, Kenneth Koltermann, and Dmitry 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*, ASPLOS '20, page 667–682, New York, NY, USA, 2020. Association for Computing Machinery.

[165] Tao Zhang, Timothy Lesch, Kenneth Koltermann, and Dmitry Evtyushkin. Stbpu: A reasonably secure branch prediction unit. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 109–123, 2022.

[166] WenTao Zhang, ZhenZhen Bao, DongDai Lin, Vincent Rijmen, BoHan Yang, and Ingrid Verbauwhede. Rectangle: a bit-slice lightweight block cipher suitable for multiple platforms. *Science China Information Sciences*, 58(12):1–15, Dec 2015.

[167] Wentao Zhang, Zhenzhen Bao, Vincent Rijmen, and Meicheng Liu. A new classification of 4-bit optimal s-boxes and its application to present, rectangle and spongent. In *Fast Software Encryption*, Gregor Leander, editor, pages 494–515, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[168] Lutan Zhao, Peinan Li, Rui Hou, Michael C Huang, Jiazhen Li, Lixin Zhang, Xuehai Qian, and Dan Meng. A lightweight isolation mechanism for secure branch predictors. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 1267–1272. IEEE, 2021.

VITA

Tao Zhang

Tao Zhang is a Ph.D. candidate in the Computer Science Department at the College of William & Mary, under the supervision of Prof. Dmitry Evtyushkin. His research focuses on microarchitecture security, side and covert channels, and secure hardware design. His Ph.D. research has been published in ASPLOS 2020 and DSN 2022. Tao has been interning at Intel since May 2022. Before joining William & Mary, he received his B.Eng. degree from North China University of Technology and his M.S. degree from Central Michigan University.