Side-Channel Attacks in RISC-V BOOM Front-End

Rutvik J Chavda

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Wenjie Xiong, Chair
Changwoo Min
Haining Wang

May 8, 2023
Blacksburg, Virginia

Keywords: Branch Predictor, Front-end, Side-channel, etc.

Copyright 2023, Rutvik J Chavda

Side-Channel Attacks in RISC-V BOOM Front-End

Rutvik J Chavda

(ABSTRACT)

The prevalence of side-channel attacks exploiting hardware vulnerabilities leads to the exfiltration of secretive data such as secret keys, which poses a significant threat to the security of modern processors. The RISC-V BOOM core is an open-source modern processor design widely utilized in research and industry. It enables experimentation with microarchitectures and memory hierarchies for optimized performance in various workloads. The RISC-V BOOM core finds application in the IoT and Embedded systems sector, where addressing side-channel attacks becomes crucial due to the significant emphasis on security.

While prior studies on BOOM mainly focus on the side-channel in the memory hierarchy such as caches or physical attacks such as power side-channel. Recently, the front-end of microprocessors, which is responsible for fetching and decoding instructions, is found to be another potential source of side-channel attacks on Intel Processors.

In this study, I present four timing-based side-channel attacks that leverage components in the front-end of BOOM. I tested the effectiveness of the attacks using a simulator and Xilinx VCU118 FPGA board. Finally, I provided possible mitigation techniques for these types of attacks to improve the overall security of modern processors. Our findings underscore the importance of identifying and addressing vulnerabilities in the front-end of modern processors, such as the BOOM core, to mitigate the risk of side-channel attacks and enhance system security.

Side-Channel Attacks in RISC-V BOOM Front-End

Rutvik J Chavda

(GENERAL AUDIENCE ABSTRACT)

In today's digital landscape, the security of modern processors is threatened by the increasing prevalence of side-channel attacks that exploit hardware vulnerabilities. These attacks are a type of security threat that allows attackers to extract sensitive information from computer systems by analyzing the physical behavior. The risk of such attacks is further amplified when multiple users or applications share the same hardware resources. Attackers can exploit the interactions and dependencies among shared resources to gather information and compromise the integrity and confidentiality of critical data.

The RISC-V BOOM core, a widely utilized modern processor design, is not immune to these side-channel attacks. This issue demands urgent attention, especially considering its deployment in data-sensitive domains such as IoT and embedded systems.

Previous studies have focused on side-channel vulnerabilities in other areas of BOOM, neglecting the front-end. However, the front-end, responsible for processing initial information, has recently emerged as another potential target for side-channel attacks. To address this, I conducted a study on the vulnerability of the RISC-V BOOM core's front-end. By conducting tests using both a software-based simulator and a physical board, I uncovered potential security threats and discussed potential techniques to mitigate these risks, thereby enhancing the overall security of modern processors. These findings underscore the significance of addressing vulnerabilities in the front-end of processors to prevent side-channel attacks and safeguard against potential malicious activities.

Dedication

This work is dedicated to my family, for their unwavering support, love and guidance throughout this journey.

Acknowledgments

Firstly, I would like to express my deepest gratitude to my advisor, Dr. Wenjie Xiong, for her exceptional guidance, and mentorship throughout my research. Her expertise, patience, and dedication have been invaluable in shaping my understanding and pushing me to achieve my best. Dr. Xiong's leadership in the BEARHW research lab has not only benefited me but also numerous other students, and I am truly grateful for her commitment to fostering our growth.

I would also want to thank the members of my thesis committee, Dr. Chanwoo Min and Dr. Haining Wang, whose valuable feedback and rigorous review have significantly enhanced the quality of this work.

A special thank you goes to my parents, who have loved me unconditionally and supported me wholeheartedly, I am eternally grateful. Your constant belief in my abilities and your unwavering encouragement have been the driving force behind my academic journey. Your sacrifices and dedication to my education have shaped me into the person I am today. I will be eternally grateful for your love and support.

Additionally, I am deeply thankful to my sister, brother-in-law, and beloved nieces for their constant support, affection, and motivation. Your presence in my life has been a source of inspiration and endless joy. I am grateful for the strong bond we share and for the love and confidence that have uplifted me throughout this journey.

Lastly, I want to give a shout-out to my amazing circle of friends. Your continuous support, understanding, and sheer presence in my life have made this journey not only meaningful and enjoyable but also downright hilarious at times. From listening to my endless rants about life's quirks on marathon phone calls to tolerating my offbeat sense of humor, you have been my rock. Your belief in me, even when I doubted myself, has provided the comic relief and strength I needed to tackle challenges and pursue my goals. I am forever grateful for the laughter, camaraderie, and unforgettable memories we've shared along the way.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Presently, the processors are typically designed to improve performance by utilizing advanced techniques such as caching and speculative execution. These techniques allow the processor to execute instructions more quickly by predicting the outcome of future instructions and executing them before they are needed. However, such hardware designs for performance could lead to security vulnerabilities.

Side-channel attacks have been a significant concern for modern processors for many years. These attacks have been found to be effective in exploiting vulnerabilities in processor designs, allowing attackers to extract sensitive information such as encryption keys [33, 49]. One of the most well-known side-channel attacks is the cache side-channel attack. This attack, affecting processors like Intel, exploits vulnerabilities in the processor caches to extract sensitive information by analyzing timing and access patterns. By carefully observing cache behavior, attackers can deduce executed instructions and potentially extract sensitive information [33].

The discovery of side-channel attacks has led to increased concern about the security of modern processors. As a result, researchers have been investigating the vulnerabilities of various processor architectures. The RISC-V BOOM (Berkeley Out-of-Order Machine) core is an open-source processor design that is gaining popularity due to its customizable and scalable architecture [10]. However, like any processor design, the BOOM core is vulnerable to side-channel attacks that can compromise its security.

One such effort to discover the vulnerability of Spectre-type attacks in BOOM had been undertaken by a group of researchers at Berkeley [23]. They provided proof-of-concept implementations for Spectre-v1 and Spectre-v2 attacks that target the L1 data cache, using cache based side-channel to extract sensitive data. Along these lines they present the need for hardware mitigations to ensure security against unauthorized access to sensitive data.

Additional research [17], demonstrates a new class of side-channel attacks exploiting the leakage of sensitive information from the processor's front-end. The front-end is a critical component in processors, responsible for fetching, decoding and delivering instructions to the rest of the pipeline. To ensure efficient processing, front-end is often equipped with multiple functional units which have their own unique timing and power signatures, which can be exploited by attackers to setup side-channels to reveal delicate activities.

This sparked off a motivation in investigating the vulnerability of the BOOM core front-end to side-channel attacks. By doing so, it could help identify potential security threats and initiate the development of countermeasures to mitigate these threats.

This research is essential because the BOOM core is used in a variety of applications, like IoT devices, embedded systems, and recently as AI/ML accelerators, where security is of utmost importance. By identifying and addressing vulnerabilities in the front-end of the BOOM core, researchers can help to ensure the security of these systems and protect against the potential risks posed by side-channel attacks.

The thesis is organized as follows. Chapter 2 provides background and related side-channel attacks. Chapter 3 explains our implementation setup. Chapter 4 discusses the overview of BOOM core and its components. Chapter 5 through 8 go over the four different side-channel attacks in the front-end of BOOM along with their respective evaluations. Chapter 9 summarizes all these attacks. Chapter 10 provides insights into potential mitigation techniques for

these attacks. Chapter 11 discusses some real-world examples the attacks can be extended to and Chapter 12 concludes.

Chapter 2

Background

2.1 Side and Covert Channels

Side and covert channels are the sneaky communication channels that can be exploited to get confidential information from a system. Side-channels and covert-channels are both methods for attackers to gain access to sensitive information in a system. Side-channels refer to unintentional leaks of information by the victim, while covert channels are intentionally created by malicious actors to bypass security measures like isolation and clandestinely transmit information as seen in Figure 2.1 [47].

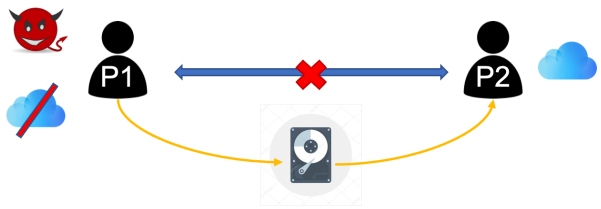


Figure 2.1: Side and Covert Channel

Generally, these channels can be classified into 4 different categories - timing, power, electromagnetic emission, and acoustic based on the way of communicating the information. Timing-based channels exploit timing differences between various operations to extract information, which led to the development of timing side-channel attacks. Out of all the categories, timing side-channels are the easiest to deploy as they only require monitoring the time taken for a computation to execute, which can often be done using standard software or

hardware tools. As a result, timing side-channels can be executed remotely without requiring physical access to the target device, making them a popular choice for attackers seeking to compromise the security of computer systems [42].

Usually, these timing side-channel attacks can further be classified into internal timing and external timing attacks. Internal timing attacks involve an attacker measuring their own execution time and using their knowledge of their own operations, such as which cache lines they accessed and the timing of these operations, to deduce information about other applications on the processor. On the other hand, external timing attacks involve an attacker measuring the execution time of the victim, such as the time it takes to encrypt data, and using this information to deduce sensitive details about the victim's operations [43].

Recently there has been a considerable growth in the number and severity of these attacks targeting very specific components of computer systems, to extract sensitive data and also circumvent any security measures deployed at kernel/application level simultaneously.

2.2 Related Conventional Timing-Based Attacks

In the last few years, a number of timing-based attacks have been discovered, which can differ both in the specific component of the processor that is vulnerable to exploitation and the side-channel used to communicate the exploited behavior. Therefore, an appropriate classification of these attacks would be based on the specific component that is exploited, as shown in Figure 2.2.

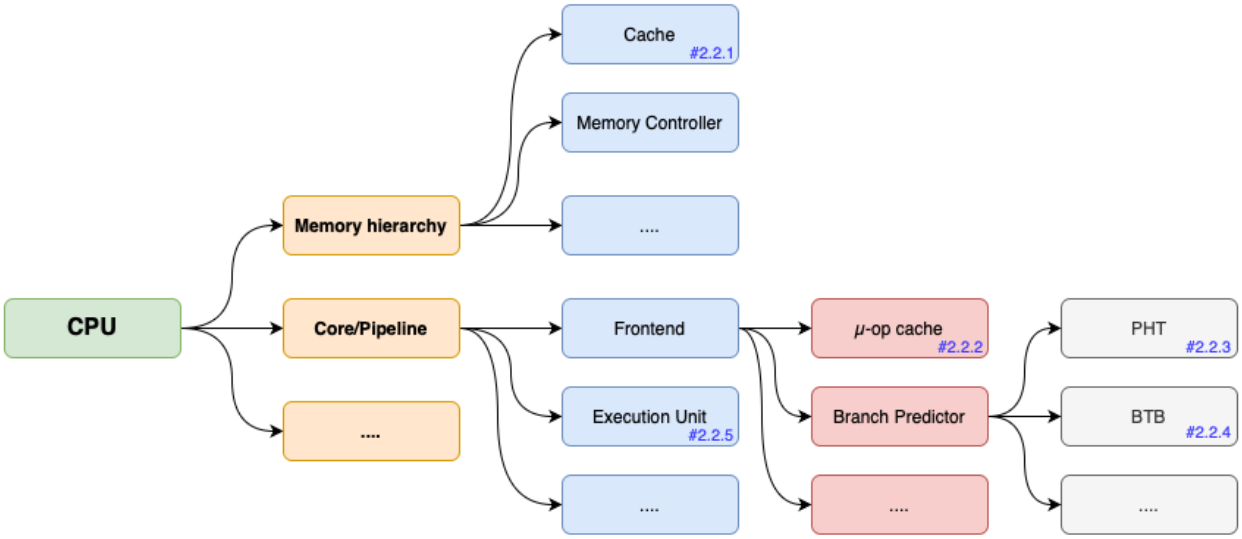


Figure 2.2: Vulnerable components in a CPU

2.2.1 Cache

PRIME+PROBE [33]. The prime+probe attack involves the attacker "priming" a cache set by loading it with their own data, then waiting for the victim process to access the same set. The attacker can then "probe" the set to see if their data has been evicted, indicating that the victim process accessed the same set. By repeatedly priming and probing different sets, the attacker can eventually reconstruct the victim's memory access pattern [33]. Based on this the attacker would know exactly which cache line to access if the victim is dealing with any secretive data.

FLUSH+RELOAD [49]. In this attack, the attacker flushes a cache set containing shared data, then waits for the victim process to access the same set. The attacker can then reload the set and measure the time it takes to reload, which indicates whether the victim accessed the set. This attack is more difficult to detect than the prime+probe attack and can be used to monitor a victim process's memory access patterns with high accuracy and low noise [49].

2.2.2 Micro-operation Cache

Micro-op (µ-op) cache attack [37]. Modern processors use a dedicated cache to store decoded micro-operations, which speeds up the processor's performance. The micro-op cache is not flushed when there is a privilege level change, meaning an attacker with kernel-level access can use it to infer information about the execution of instructions in user-mode programs. By carefully crafting instructions in the kernel to prime the micro-op cache, an attacker can measure the timing of specific micro-operations that are dependent on a secret value [37]. This allows the attacker to extract the secret value by analyzing which micro-operations are present in the micro-op cache, even though the cache is not intended to be visible to software.

Leaky front-end [17]. In modern processors, the front-end is the component responsible for fetching and decoding instructions. It prepares the instructions for execution and sends them to the rest of the processor. The front-end is critical for the performance of the processor, but it is vulnerable to attacks that exploit the way it handles speculative execution. These vulnerabilities are caused by the multiple paths that micro-operations can take through the front-end, including the Micro-Instruction Translation Engine (MITE), the Decode Stream Buffer (DSB), and the Loop Stream Detector (LSD). Each path has unique timing and power signatures that can be exploited to create side-channel and covert-channel attacks. The switching between different paths leads to observable timing or power differences that attackers can exploit to deduce sensitive information [17]. Furthermore, since these components are shared between hardware threads, two separate threads can influence each other, and timing or power differences can reveal activity on the other thread .

2.2.3 Pattern History Table (PHT)

Branchscope [20]. Branchscope leverages the directional branch predictor to spill out sensitive information [20]. A directional branch predictor like PHT is a hardware component that stores the prediction of a given branch. This property is exploited by Branchscope to forecast the branch that might leak sensitive information. The attacker begins the attack by first identifying the sensitive branch on the victim's side and then using a series of conditional branch statements to compel the victim program's branch to be predicted a certain way and then quickly swooping in and studying the direction of the predictor to retrieve the secret key.

BranchSpec [15]. BranchSpec takes advantage of the same behavior of PHT by repeatedly executing a set of specially crafted branch instructions that are designed to leak information about a victim's system. By observing the timing differences between correctly and incorrectly predicted branches, an attacker can infer sensitive information [15]. The attacks include a side-channel and a covert-channel that perturbs the branch pattern history structure. These attacks can take advantage of simpler code patterns, potentially making the impact of exploitation even more severe.

2.2.4 Branch-Target-Buffer (BTB)

Jump over ASLR [19]. Evtyushkin et al. [19] demonstrate a side-channel attack through the branch target buffer (BTB) that can be used to bypass ASLR (Address Space Layout Randomization) in current computing systems. ASLR is a security measure that randomizes the memory address space of essential data structures or executables in a system to prevent attacks such as buffer overflow. However, the BTB, which stores target addresses of recent branch instructions, can be used by attackers to predict the location of critical system

components. By repeatedly executing a sequence of instructions that trigger a specific conditional branch instruction and analyzing the access time to the target memory location, the attacker can infer the memory layout and bypass ASLR. This demonstrates the vulnerability of the BTB as a side-channel attack vector.

2.2.5 Execution Units

PORTSMASH [8]. It is an innovative side-channel attack that exploits an inherent component of modern processors, specifically targeting Intel Hyper-Threading technology. Simultaneous Multithreading (SMT) architectures, offer a broader attack surface for side-channel attackers, exposing more microarchitecture components per physical core compared to cross-core attacks. PORTSMASH utilizes timing information derived from port contention to the execution units, thus targeting a non-persistent shared hardware resource and extracts sensitive information [8].

2.3 Speculative Execution Attacks

In addition, side-channel can also be utilized to carry out speculative execution-based attacks. Speculative execution is an optimization technique wherein the CPU predicts the instructions that might be used in the future and starts executing them before even they are needed.

Spectre [28]. Another notorious attack is Spectre, which exploits this speculative execution behavior of modern processors to leak sensitive information [28]. Spectre attack trains the predictor to let an out-of-bounds check bypass a conditional statement and put the sensitive data in transient microarchitecture states, and then use a covert channel (e.g., covert channel in caches) to receive the data. It uses this vulnerability by deceiving the CPU into executing

sensitive instructions that shouldn't have been executed in the first place.

SpectreRSB [29]. The RSB is a data structure to track return addresses of function calls. This allows the processor to speculatively execute instructions after the function call, improving performance. The attacker exploits the RSB by tricking the victim's processor into speculatively executing instructions that use the RSB to load sensitive information into an array and then probing the elements allows the attacker to infer the data [29].

Meltdown [31]. Meltdown is another security vulnerability that exploits the speculative execution behavior. By tricking the processor into speculatively executing instructions that access kernel memory, Meltdown allows an attacker to indirectly infer the content of privileged memory. Through timing measurements and cache side-channel effects, the attacker can deduce sensitive data stored in the kernel memory, even though it should be inaccessible from user space [31]. This way, the Meltdown attack can retrieve sensitive information from the kernel memory without legitimate access rights.

Even though Meltdown and Spectre exploit speculative execution, Spectre focuses on accessing arbitrary memory locations, while Meltdown specifically targets the ability to read kernel memory from user space.

In response to the threat of speculative attacks on modern processors like Intel/AMD, significant efforts have been made to implement effective mitigations. One approach involves the implementation of microcode updates and firmware patches. These updates modify the processor's behavior and prediction algorithms, specifically targeting vulnerabilities exploited by speculative execution attacks.

Features like Indirect Branch Restricted Speculation (IBRS) and Indirect Branch Predictor Barrier (IBPB) are implemented through software control mechanisms to selectively restrict the indirect branch predictor. IBRS restricts the speculative execution of indirect branches,

minimizing the risk of attacks that rely on branch prediction [4]. IBPB acts as a barrier by flushing outdated or potentially malicious branch predictions, preventing their utilization during speculative execution [3]. Another approach also discussed in section 10.1 is reconstructing the kernel without indirect jump and use 'retpolines' instead [5, 41].

Chapter 3

Implementation

To test the effectiveness of our side-channel attacks on the BOOM core, I have utilized both FPGA board and a simulation software. These two platforms have allowed us to evaluate the performance of our attacks under a range of conditions and identify areas for improvement in the attack algorithm. Figure 3.1 illustrates the implementation flow of the setup, providing an overview of how the system is structured and organized.

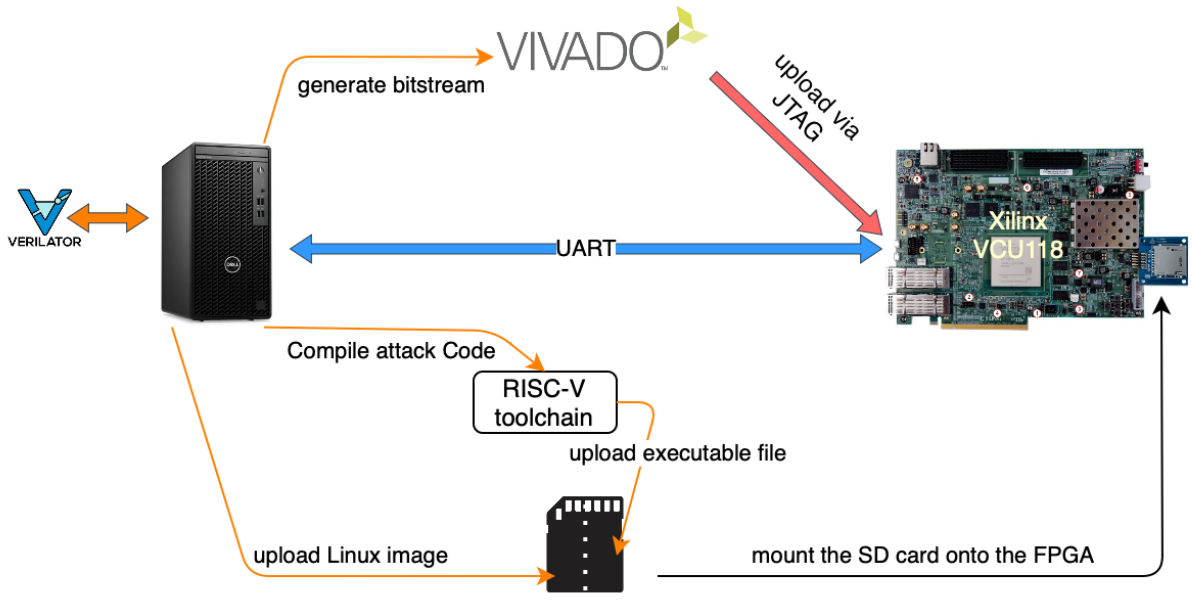


Figure 3.1: Implementation Setup

3.1 Simulator Setup

To perform side-channel attacks on the BOOM core, I used Verilator to simulate the core's hardware design on the host machine. I modified the configuration file for the BOOM core to enable and disable various components for testing under different scenarios. I then compiled the core with Verilator, providing the modified configuration file and the necessary side-channel attack binary file during the build process. Once the simulation was running, I used the output files generated by Verilator to analyze the effectiveness of the side-channel attack, utilizing HPM (Hardware Performance Monitor) counters built-in the BOOM core to gather necessary performance data.

3.2 FPGA Setup

In order to test our side-channel attacks on an actual hardware I use the Xilinx VCU118 FPGA board that is running a stripped-down version of Linux on an instance of 1-wide Small BOOMv3 (`SonicBOOM`) core.

The default harness that runs the `SmallBOOM` core on the FPGA was modified to enable Linux to run on the board, along with UART and SPI SDCard extensions for communication with the host machine via UART.

I expanded the harness to adopt configurations of the `SmallBOOM` core that were tailored to our specific use-case. This allowed us to enable or disable specific components as needed, enabling us to thoroughly test our side-channel attacks in different scenarios. To generate bitstreams for these configurations, I used Vivado, which was then programmed onto the FPGA. The Linux image was loaded from the SDCard on boot-up.

Lastly, I used the *riscv-gnu* toolchain on our host machine to generate the binary files for

our side-channel attacks. These files are loaded onto the SDCard and executed to gather the necessary performance data about the effectiveness of the attack.

3.2.1 Limitations and Challenges

One of the main limitations of our setup was not able to use JTAG for debugging purposes. The default harness provided in the Chipyard framework for Xilinx VCU118 is only designed to support UART. I tried to incorporate a JTAG module into the harness, but was unsuccessful in doing so. I had to resort to manually uploading our designs onto the SDCard and then using UART to view print statements.

Initially, I was on the track to use a wrapper CONFIG class provided by Raphael Klink [18] that included the necessary module instances for JTAG and UART. To upload and debug our designs on the FPGA, I utilized an external JTAG debugger tool called the HS2, in conjunction with the *riscv-gdb*. However, I encountered my first obstacle when attempting to run bare-metal programs, as nothing was displayed on the UART terminal. I also attempted to upload a Linux image through the debugger, hoping to directly run the attacks from there, but I was unsuccessful in getting the Linux image to load.

Chapter 4

BOOM Overview

BOOM is a RISC-V Instruction Set Architecture (ISA)-based microprocessor architecture. It is an open-source architecture designed by the UC Berkeley, with the goal of providing a high-performance yet simple RISC-based processor design. The BOOM pipeline is a super-scalar out-of-order pipeline that can handle up to six instructions per cycle and employs a dynamic instruction scheduler to allow instructions to be processed out of order, improving processor throughput. It has the potential to lower utilization while improving overall performance. The BOOM pipeline is divided into his three main phases (illustrated in Figure 4.1): Front-end, Execution Engine, and Retirement Phase [1].

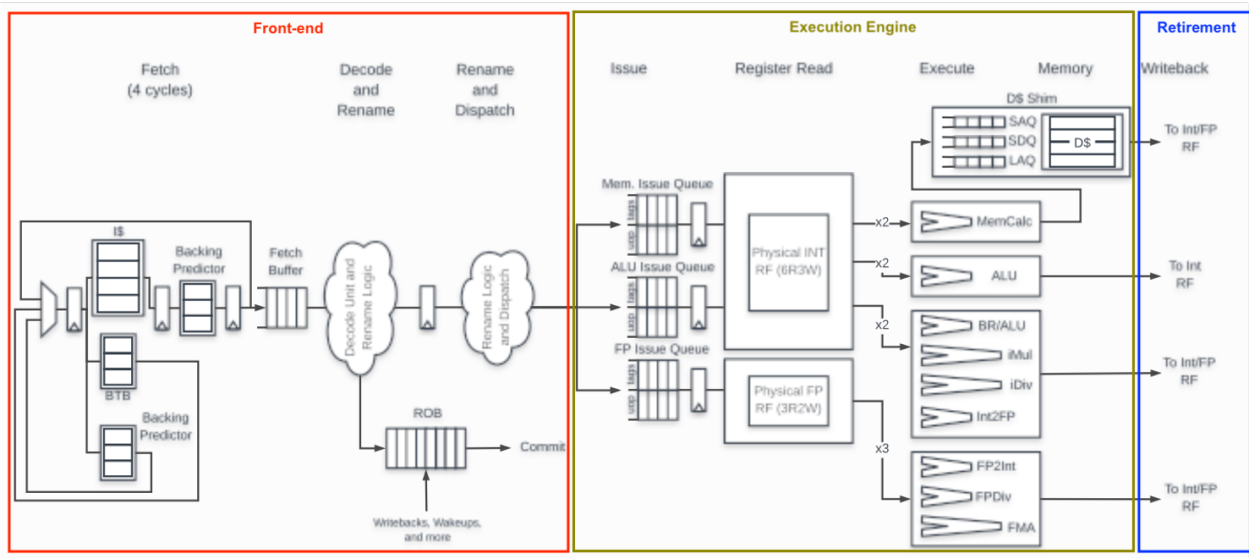


Figure 4.1: The BOOM pipeline [1]

4.1 Use Cases of BOOM Core

The purpose of designing BOOM was to create a prototype processor that can be used as a baseline for future micro-architectural studies of out-of-order processors. The main objective was to develop a readable, open-source implementation that can be utilized in education, research, and industry [10].

RISC-V BOOM core has the potential to advance both industry and research fields by providing a flexible and customizable platform for creating processors that are optimized for specific applications. For instance, in the research sector, BOOM core can be utilized to experiment with new microarchitectures and memory hierarchies to optimize performance for particular workloads. The open-source nature of BOOM core enables researchers to collaborate and share their findings with others, resulting in innovative solutions to improve computer architecture.

In the industry sector, RISC-V BOOM core offers several benefits for industry applications, including IoT devices, cloud computing, edge computing, and AI/ML accelerators. BOOM core can be optimized for IoT devices to enable small and efficient processors with low power consumption. For cloud computing, BOOM core can be customized to support virtualization, hardware acceleration, and specialized instructions for data processing. In edge computing, BOOM core can be tailored to include machine learning accelerators and specialized instructions for sensor data processing. And for AI/ML accelerators, BOOM core can be customized to include specialized processing units for efficient and high-performance machine learning applications. By utilizing the flexibility and customization options of BOOM core, companies can create processors that are optimized for their specific applications, resulting in improved performance, energy efficiency, and user experience.

4.2 Related RISC-V Cores

There are currently 107 RISC-V core designs, SoC and other platforms listed on the RISC-V project overview [22], each with its own unique features and optimizations for different use cases. These cores range from simple in-order pipelines optimized for low power consumption to complex out-of-order pipelines designed for high-performance computing and machine learning workloads.

Below are some of the most distinguished RISC-V core apart from BOOM:

- Rocket: The Rocket core is an open-source RISC-V implementation that serves as a base for designing custom SoCs. It supports RV32I and RV64I instruction sets and includes a 5-stage in-order pipeline with support for instruction and data caches, branch prediction, and interrupt handling. Rocket is intended for use in embedded systems, IoT devices, and other low-power applications [9].

- CVA6 (Ariane): The CVA6 core is a 6-stage pipeline RISC-V core developed at ETH Zurich that supports RV64GC instruction set. The additional pipeline stage is for Program Counter (PC). It is intended for use in high-performance computing, data centers, and AI/ML accelerators [50].

- Shakti-C: The Shakti-C is a RISC-V core developed by IIT Madras. It is designed to be a high-performance and configurable RISC-V processor that supports both RV32I and RV64I ISAs. It is optimized for high throughput and power efficiency, making it suitable for use in cloud computing and high-performance computing applications [21].

4.3 Front-end in BOOM

The BOOM pipeline front-end fetches instructions from memory, decodes them, and sends them to the execution engine. The front-end is made up of several submodules that work together to fetch, decode, and dispatch instructions [1].

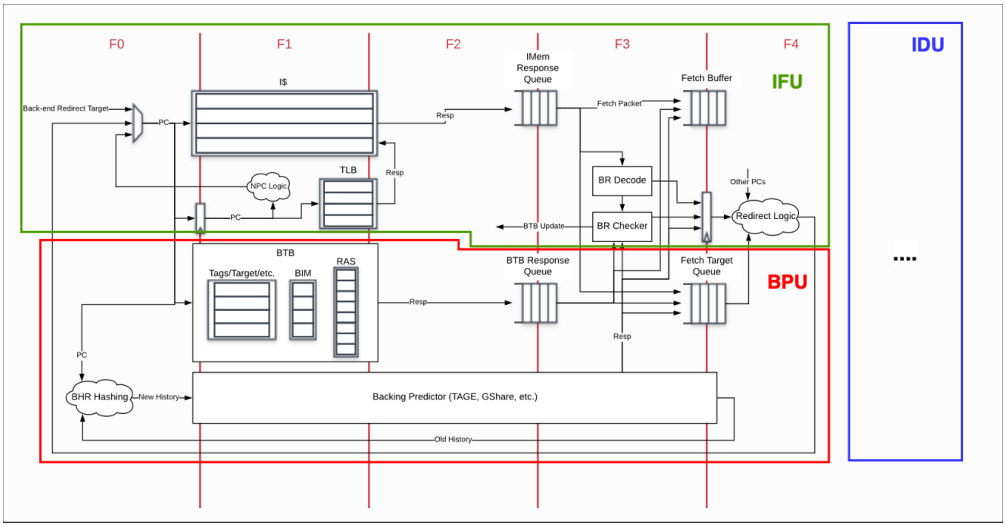


Figure 4.2: The BOOM Front-end [1]

From Figure 4.2, the first submodule of the front-end is the Instruction Fetch Unit (IFU). The IFU is responsible for fetching instructions from memory and sending them to the next stage of the pipeline. The IFU is designed to handle both sequential and non-sequential instruction fetches. In other words, branch prediction may be used to retrieve instructions from either contiguous or non-contiguous memory locations.

The Branch Prediction Unit (BPU) is the second submodule of the front-end. The BPU predicts whether a branch instruction will be taken and provides this information to the IFU to fetch the correct instruction. BOOM uses a two-level adaptive predictor that can predict both local and global branches. The BPU is critical for improving processor performance by accurately predicting branch instructions, thereby reducing pipeline stalls and enabling better utilization of the processor's resources.

Finally, the instruction decode unit (IDU) is the front-end's third sub-module. The IDU is in charge of decoding the instructions retrieved by the IFU and creating micro-operations (µ-ops) that the execution engine may perform. It is also responsible for efficiently allocating the required resources for each instruction and, if a resource is unavailable, it can cause pipeline stalls to ensure proper execution.

4.4 Branch Prediction

As discussed earlier, the BPU is in charge of predicting the result of conditional branches. A conditional branch is an instruction that causes the program to go to a different section of code based on the value of a specified register or memory address. The BPUs task is to correctly predict the outcome of the branch, in order to speculatively execute instructions after the branch without anticipating the actual outcome of the branch [1].

BOOM's BPU is made up of several components, each of which serves a distinct purpose. These components include:

1. Branch Target Buffer (BTB): The BTB retains the addresses of previously executed branches as well as the addresses of their targets. When a branch instruction is detected, the BPU searches the BTB to see if the branch was previously taken. In that case the predicted destination address is taken from his BTB.

2. Bi-Modal (BIM) Table: The BIM table, sometimes referred to as the Pattern History Table (PHT) is a data structure that stores the outcomes of recent branches. It records the history of the last several branches and their predictions. By analyzing this pattern of recent results, the BPU uses the information stored in the Bi-Modal table to anticipate the outcome of the current branch. Som

3. Branch History Table (BHT): A table that tracks the results of all branches. Unlike BIM, BHT stores the results of all branches, not just the last branch. Each entry in the BHT corresponds to a specific branch instruction and retains the history of that branch. BHT is used to predict the outcome of a branch based on the history of outcomes for that given branch.

4. Two-level Adaptive training: TAgged GEometric (TAGE) length predictor is an advanced branch predictor that makes even more accurate predictions by combining the local and global histories. The TAGE predictor is made up of several tables, each with a distinct history length. Based on the history of the recent branches, the BPU utilizes these tables to predict the accurate outcome.

5. Return Address Stack (RAS): As the name suggests it's a stack that holds the return addresses of executed call instructions. The BPU pops the topmost address from the RAS whenever a 'ret' instruction is encountered, to predict the return target.

4.5 Levels of Branch Prediction in BOOM

BOOM employs two-levels of branch prediction. The first level is a speedy Next-Line Predictor, while the second level is a more intricate Backing Predictor that, although slower, is more accurate in its predictions. By utilizing both NLP and BPD, BOOM is able to make more informed decisions on which path to take when executing code, ultimately improving its performance and efficiency.

4.5.1 Next-Line Predictor (NLP)

The NLP is designed to take the current Program Counter (PC) value to fetch instructions and predicts where to fetch the next instruction in the upcoming cycle. If the prediction is correct, there is no need to stall the pipeline. But if there is a mispredict detected further down the line, then a request is sent to the front-end to follow the new instruction path [35].

It is a combination of a fully associative BTB, a Bi-Modal Table (BIM), and a RAS that work conjointly to make a fast reasonably accurate prediction. The BIM is nothing but a branch history table, with a 2-bit saturating counter representing the likely hood of that branch being taken or not [1].

When the Fetch PC comes across a branch instruction, it uses a tag match to locate a uniquely matched entry in the BTB. If a matching BTB entry is found, a prediction is made in conjunction with RAS as to whether the instruction is a jump, branch, or return and also the responsible instruction in the Fetch Packet. This instruction responsible for control flow prediction is identified by the branch index (bidx) bits in the BTB entry. The BIM table is then requested to assess the prediction made by BTB was a taken or not taken branch. Considering the BTB entry is a return instruction, the RAS supplies the predicted return PC as the next Fetch PC. BIM in this case and during an unconditional jump is not consulted for a decision [1]. The rundown of NLP is illustrated in Figure 4.3.

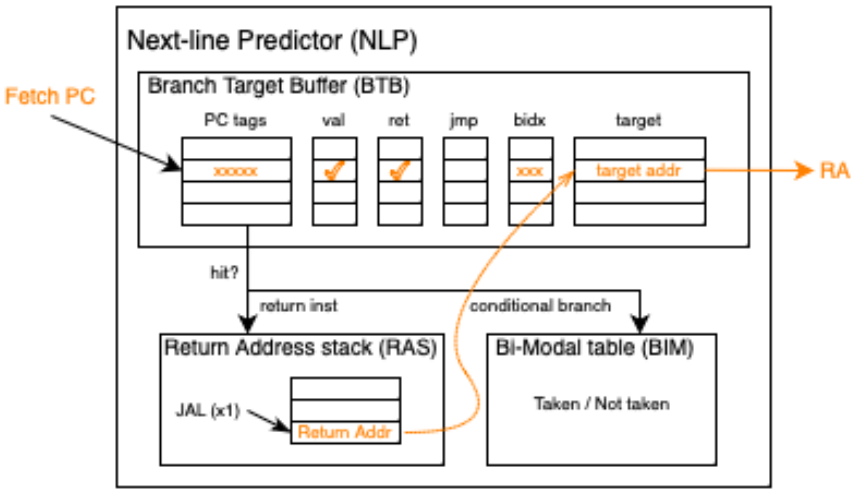


Figure 4.3: The Next-Line Predictor (NLP) in BOOM

4.5.2 Backing Predictor (BPD)

The Backing predictor is a component of a 2-level branch predictor that makes prediction by combining global and local histories. Global history is the history of previous branches spanning all program executions. While local history is a record of former branches within a specific region of a program. The global history is stored in a register called the Global History Register (GHR), which is used to index into the backing predictor table [1].

The backing predictor table contains entries that correspond to particular combinations of global history bits, and each entry stores information about the outcome of the last branch instruction encountered with that particular combination of global history bits. When a new branch instruction is encountered, the backing predictor uses the current value of the GHR to index into the backing predictor table, retrieves the corresponding entry, and uses the stored information to make a prediction about the outcome of the current branch [1].

The GHR is updated with the outcome of the current branch instruction, and the least

significant bit of the GHR is used to determine the next index into the backing predictor table. This allows the backing predictor to adapt to changes in the behavior of the program over time. There are several types of two-level branch predictors that use different techniques to make predictions based on global history [1]. Two such predictors are GShare and TAGE.

The GShare predictor is a type of two-level branch predictor that uses a hash function to map the global history to an index in a table of predictor entries. The predictor entries store a prediction bit and a counter that is incremented or decremented based on the actual outcome of the branch. The prediction bit is used to make the actual prediction, and the counter is used to update the prediction accuracy over time [1].

The TAGE (TAgged GEometric) predictor is a type of two-level branch predictor that uses tagged geometric history lengths to predict branches. It maintains multiple prediction tables with different history lengths, and selects between them based on the recent prediction accuracy. Each table entry in TAGE stores a prediction bit and a "tag" that is used to match the entry with a particular branch instruction. The prediction bit is used to make the actual prediction, and the tag is used to determine which entry in the table to update based on the outcome of the current branch [1].

The TAGE predictor also uses a technique called "history-based indexing" to index the internal tables. It maps branch instructions to different locations internally based on the history of previous branch outcomes, and not solely based on the address of the instruction. The branch history is a sequence of past branch outcomes, which is used to calculate an index into each of the internal tables. A hash function takes into account both the branch address and the branch history to calculate the index. The resulting index is used to access the corresponding entries in the internal tables, which contain information about the branch outcome and the predicted target address [1]. The overview of the TAGE predictor is depicted in Figure 4.4.

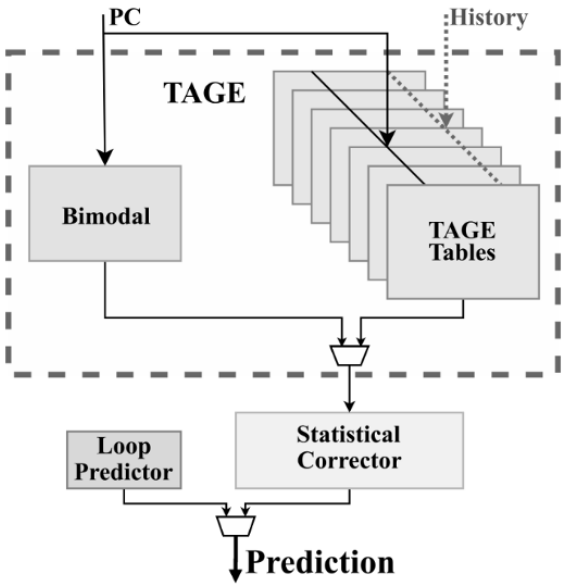


Figure 4.4: An abstract representation of TAGE predictor [45]

The loop predictor works in tandem with the TAGE branch predictor to improve the accuracy of predictions for loops that have a constant number of iterations. The TAGE predictor demonstrates remarkable accuracy in predicting the loop behavior when the control flow remains consistent. However, if the control flow becomes erratic, the TAGE predictor may struggle to accurately predict the loop's termination [39].

The loop predictor plays a vital role in identifying regular loops with a fixed iteration count. It provides global predictions a branch is identified as a loop with a constant number of iterations. These predictions gain higher confidence when the loop has been executed multiple times with the same iteration count. By incorporating the loop predictor alongside the TAGE branch predictor, more precise and dependable predictions can be achieved, particularly for loops that exhibit consistent behavior [39]. Detailed explanation of the same is given in Section 8.1.

Chapter 5

BRAD-v1 Attack

In this chapter, I present BRAD-v1 attack, which leak the branch direction of a victim program. I discuss the design goal and overview (section 5.1), detailed design (section 5.2), and finally evaluate and verify (section 5.3) the BRAD-v1 (BRAnch Direction) attack.

5.1 Goals and Overview

Learning from the previous sections, we understand that modern processors utilize branch predictors and speculative execution to optimize performance. The objective is to leverage the prediction outcome for attacks. To achieve this, the attacker must construct the attack in a manner that causes the victim's execution change the state of a branch predictor. Consequently, this would allow confidential information to be disclosed through a side-channel to the attacker.

The threat model is that there exist two distinct parties, namely, the victim and the spy. Only the victim code block has the capability to access the sensitive information. Meanwhile, the spy seeks to deduce the sensitive information, without directly accessing the secret. Moreover, this attack presumes that the spy can invoke the victim code block to execute the intended vulnerable operation at any given moment.

Before I delve into the specifics of the attack, I shall provide some background information

and a high-level overview of the attack. The attack generally proceeds as follows:

- *Stage 1: Victim Trains the BTB entry.* The victim block is invoked multiple times to train a specific BTB entry to be either taken or not-taken.

- *Stage 2: Spy Times the BTB access.* The spy block executes a branch instruction targeting the same BTB entry as the victim and simultaneously times its own execution to observe the victim's predicted outcome.

The primary goal is to cause a collision between the branches of the victim and the spy, with the spy code causing the collision. These collisions enable the attacker to determine the direction of the victim's branch by observing the effect of that branch on the accuracy of prediction of an attacker's probing branches in *stage 2*. BTB is a memory component that retains the destination addresses of recently executed branch instructions, as introduced in Section 4.2. This enables quick access to the instructions starting from the target address during the next cycle via a BTB lookup. However, since multiple applications on the same core use the BTB, there is a possibility of information leakage between them through the BTB side-channel. Creating collisions in the BTB between the branches of two processes is more accessible if the BTB indexing is strictly determined by the instruction address, as in the case of a 1-level predictor. Thus, the BRAD attack is based on:

1. Creating Collisions: The attack relies on establishing collisions within the predictor, which is significantly more straightforward when a simple 1-level predictor is used, rather than a more sophisticated predictor like TAGE or GShare.

2. Timing the colliding branch: Once a collision in the BTB is forced, the attacker must still be able to time its execution and infer the direction of the victim's branch.

5.2 Implementation of BRAD-v1

5.2.1 Enabling NLP

The 1-wide SonicBOOM core has the state of the art TAGE-L branch predictor enabled by default [51]. As discussed in section 5.1, the first step in creating a collision is to disable the TAGE-L predictor and enable a 1-level predictor i.e., the NLP. Using the information from section 4.3.1, I wrote a config encompassing the BTB and BIM to act as the branch predictor unit in the front-end.

5.2.2 Attack Run-down

Enabling even the simplest branch predictor may not guarantee that an attacker's branch will coincide with the victim's in the Branch Target Buffer (BTB), and traversing all BTB entries to check for a hit may not be feasible. In order to establish a BTB-based side-channel, three requirements must be met.

1. The victim must fill a BTB entry by executing a branch instruction.

2. The state of the BTB must affect the execution time of the spy, who is also running on the same core. This condition is fulfilled when both code blocks use the same BTB entry, which may or may not contain different target addresses.

3. The spy must be capable of detecting the impact on its execution by performing time measurements. To address this, one way is to keep the conditional branch statement the same for both the victim and the spy. The difference is that the output of the branch depends on the secret value when the victim accesses it, and an arbitrary value when the spy accesses it.

To exploit this vulnerability, the attacker first trains the BIM table associated with the conditional branch entry in the BTB by calling the victim function 2 times, which accesses the secret data. By calling the function twice, the BHT entry corresponding to the branch is trained to be in either the taken or not-taken state. Because the counter is limited to 2 bits, it only takes two iterations to reach the maximum or minimum values.

Afterwards, the spy function is called to access a different data that is also affected by the same conditional branch. If the branch was accessed before by the victim block and the outcome was predicted by the core, it would result in a lower execution time. By measuring the execution time of the spy block, the attacker can deduce the value of the secret data. This summarizes one attack iteration, and multiple iterations may be required to account for any interference or extraneous processes that could affect the cycle counts, in order to obtain an exact and precise output of the secret value accessed across all of the attack rounds. The prototype code for this attack is listed in Figure 5.1.

5.2.3 Measuring Execution Time in BOOM

As mentioned in Section 5.1, the second-most important aspect of BRAD-v1 is to time the BTB access for the attacker, to then infer the data based on the difference in access times. Authors et al. [23] provide a proof-of-concept for speculative-style attacks in BOOM wherein they use a special instruction defined in the RISCV ISA to read the number of cycles. The instruction is '*csrr*' that stands for "control and status register read". It is used to read the value of a control and status register (CSR) in the RISC-V processor. More specifically, the '*csrr*' instruction is used to read the value of the cycle counter CSR (counter 0xc00), which counts the number of cycles since the processor was started.

```
1  uint8_t sec_data[] = {1,0,0,1,...};
2  uint8_t array1[1] = {1}; // or 0
3
4  // Function with conditional branch
5  void condBranch(uint8_t* addr) {
6      if(*addr)
7          asm("nop, nop\n");
8      else
9          asm("addi t1, zero, 2\n");
10 }
11
12 void victim_f(uint8_t idx) {
13     condBranch(&sec_data[idx]);
14 }
15 void spy_f(uint8_t idx) {
16     condBranch(&array1[idx]);
17 }
18
19 int main(void) {
20     for(int i=0; i<SECRET_DATA_LEN; i++) {
21         for(int k=0; k<ATTACK_ROUNDS; k++) {
22             // Training BIM entry by calling victim twice
23             for(int j=0; j<2; j++) {
24                 victim_f(i);
25             }
26             start = time();
27             spy_f(0); // calling spy
28             end = time();
29             store_collision_data(end-start);
30         }
31     }
32 }
```

(a)

```
...
10176: beqz  a5,1017e
10178: nop # *addr = 1
1017a: nop
1017c: j   10182
1017e: li t1,2 # *addr = 0
10182: nop # end of if
...
```

(b)

Figure 5.1: Prototype code for BRAD-v1 attack (a) and Disassembly of if-statement (b)

5.3 Evaluation and Verification

5.3.1 Simulator

Before actually running our attack code on the FPGA, I evaluated the attack on a simulator, Verilator. Verilator is a software application utilized for compiling Verilog and SystemVerilog

sources into highly optimized and optionally multithreaded cycle-accurate C++ or SystemC code. The transformed modules can then be instantiated and integrated into a C++ or SystemC testbench to serve verification and modeling purpose [30].

The BRAD-v1 attack relies on exploiting the timing differences between correct branch prediction and incorrect branch prediction in order to extract sensitive information from the victim code. However, when I performed the attack using Verilator, I noticed that the timing accesses were the same even when there was a colliding branch instruction between the spy and the victim blocks.

One reason why the attack was unsuccessful is because Verilator is a cycle-accurate software simulator for hardware description languages. However, it does not simulate time at a granular level within a single clock cycle and does not simulate the exact timing of circuits. Rather, it typically evaluates the circuit state once per clock cycle, making it impossible to observe any abnormalities that may occur within a specific interval. Moreover, it does not support timed signal delays. Therefore, Verilator may not accurately model the precise timing behavior of the underlying hardware, such as the pipeline stages and memory hierarchy. This results in different timing behavior and makes it more difficult or impossible to carry out this attack on a simulator.

5.3.2 FPGA

Running the attack on the FPGA, gave some promising results as the timing behavior was more accurately modeled considering its on the actual hardware itself.

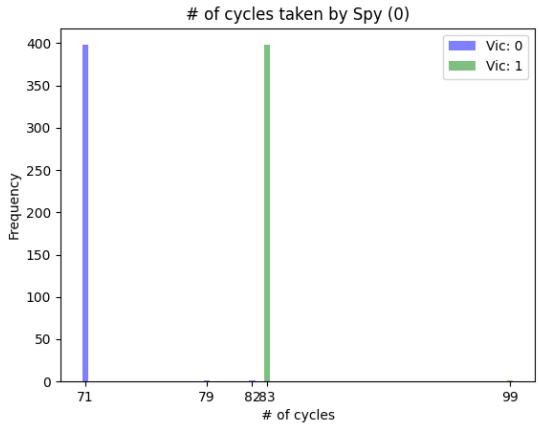
Before executing the attack on the FPGA, the base predictor (NLP) configuration bitstream is uploaded onto the FPGA along with the attack code on the SDCard. Once the Linux boots up, the attack binary file is ran multiple times to account for any noise in measurement

of cycle counts, and the output is appended to a text file. This text file is then parsed to generate the necessary plots.

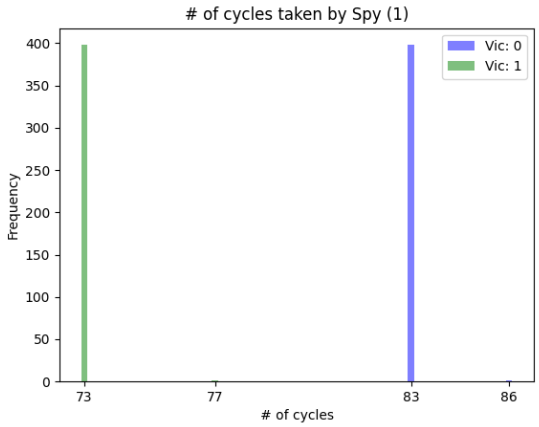
There are two distinct situations illustrated in Figure 5.2. One scenario corresponds to when the spy value is equal to '0' (Figure 5.2a), and the other when it is equal to '1' (Figure 5.2b). Upon examining the graphs, it is evident that when the victim accesses a '1', it leads to the conditional branch **not** being taken (see Figure 5.1b). Therefore, if the spy value matches that of the victim (i.e., '1'), the execution time will be lower than if the values were different. To ensure the consistency of our findings, I repeated the same attack with the spy accessing a '0' and the victim accessing secret data that alternates between '0' or '1'.

To keep things simple, I have used an 8-bit secret data array that contains an even distribution of random 0's and 1's. The green bar shows the results when the targeted block accesses a **1** from the secret array, while the blue bar represents a **0**. In order to minimize any disturbances, I took about 100 measurements during each attack round before saving the output to a text file. Since the data remained stable after the first run, I did not feel the need to record data from multiple runs. These results have also been organized in tabular form in Figure Figure 5.2c.

Coming back to the required training rounds for the BIM entry associated with the victim block execution, it is noteworthy that each BIM entry represents the branch condition using just 2 bits. The transition from one extreme state to the taken/not-taken state only requires 2 iterations, which aligns with practical observations as shown in Figure 5.3. Initially, when the victim block is called only once before the spy block executes, the behavior is inconclusive. However, with an increase in the number of iterations for the victim block execution, the results become more convincing.



(a) Spy accessing value 0

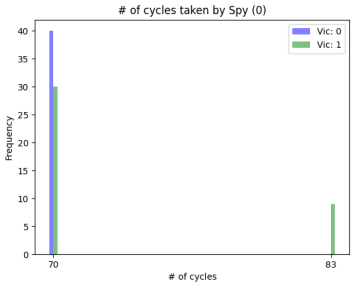


(b) Spy accessing value 1

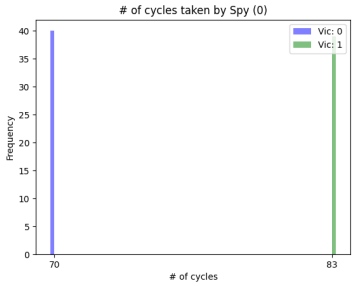
	Spy 0	Spy 1
Victim 0	71.048	83.007
Victim 1	83.040	73.01

(c)

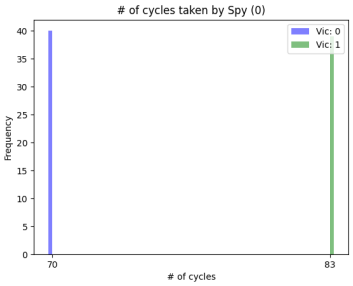
Figure 5.2: Results for BRAD-v1 attack on FPGA



(a) victim runs 1 time



(b) victim runs 2 times



(c) victim runs 3 times

Figure 5.3: BRAD-v1 attack with multiple victim block executions

Chapter 6

BRAD-v2 Attack

The attack from the previous section was limited to 1-level predictor which is the NLP, but [23] shares that there is a way this attack can be extended to target advanced branch predictors like TAGE.

Kocher et al. [28] mention that Spectre-v1 attack works by exploiting the branch predictor's training algorithm. The branch predictor uses historical information about the program's behavior to predict which branch it will take in the future. This information includes the program's execution history, the outcome of previous branches, and the program's control flow.

The attacker can train the branch predictor to make an incorrect prediction by providing it with a sequence of inputs that cause the program to take a specific branch. Once the branch predictor has been trained, the attacker can supply a new input that causes the program to take a different branch. The branch predictor will still predict that the original branch will be taken and will speculatively execute the corresponding instructions.

The *Bounds Check Bypass attack* (Spectre-v1) demonstrated here [23] works by training the TAGE branch predictor to repeatedly execute a specific input that causes the victim to take a certain branch. The attacker can then use the branch predictor to predict the outcome of the conditional branch for a different input.

During the attack phase, the attacker supplies a value to the victim that fails a bounds check

in the victim's code, but that the attacker knows will cause the victim's conditional branch to predict that a certain direction will be taken. The attacker then speculatively executes code that reads a secret value from the victim's memory location based on this prediction.

The attacker uses a cache side-channel attack to extract the secret value. Specifically, the attacker creates an "attacker array" that is populated with data that depends on the secret value. The attacker then uses the secret value as an index into the attacker array, causing a cache hit if the secret value matches the data in the array. By timing the cache hits, the attacker deduces the value of the secret. The attacker repeats this process with different inputs until they have successfully extracted the entire secret value.

This style of attack can be extended to instead create a side-channel attack on the BTB. The attacker would first identify a vulnerable piece of code in the target program that uses a conditional branch instruction, then use a similar process to train the BTB predictor to predict the outcome of the conditional branch instruction for a specific input.

During the attack phase, the attacker would supply an input that causes the conditional branch instruction to take the opposite direction than the one predicted by the BTB predictor. However, the predictor would still execute the corresponding instructions for the predicted direction, allowing the attacker to perform a side-channel attack. The secret data can then be inferred based on the access time for each data value.

6.1 Implementation of BRAD-v2

Designing BRAD-v2 requires a bit of tweaking to the code from the earlier attack. Previously I was directly accessing the target address from the BTB whose prediction depended on the BIM table output. Since, training the BIM table entry for a specific conditional branch

```
uint8_t sec_data[] = {1,0,0,1,...};
uint8_t array1[11] = {[0 ... 9] = 0,1}; // or 1

// Function with conditional branch
void branch(uint8_t* addr) {
    if(!val) {
        __asm__ __volatile__("nop\n"
               "nop\n");
    }
    else {
        __asm__ __volatile__("nop\n"
               "nop\n");
    }
}
int main(void) {
    uint64_t attackIdx = (uint64_t)(sec_data - array1);
    uint64_t passInIdx, randIdx;

    for(int i=0; i<SECRET_DATA_LEN; i++) {
        for(int k=0; k<ATTACK_SAME_ROUNDS; k++) {
            for(int j=TRAIN_TIMES; j>=0; j--) {
                randIdx = k % 10;
                passInIdx = ((j % (TRAIN_TIMES+1)) - 1) & ~0xFFFF;
                passInIdx = (passInIdx | (passInIdx >> 16));
                // select randIdx or attackIdx based on the round
                passInIdx = randIdx ^ (passInIdx & (attackIdx ^ randIdx));

                start = time();
                branch(array1[passInIdx]); // Victim
                end = time();
                store_training_data(end - start);
            }
        }
        attackIdx++;
    }
}
```

Figure 6.1: Prototype code for BRAD-v2 attack

doesn't involve that much complexity I was easily able to create a collision for the victim and the spy block on the same branch.

With predictors like TAGE, it is not that simple. From section 4.3.2, we know that TAGE predictor maps branch instructions to different locations internally based on the history of

previous branch outcomes, and not simply based on the address of the instruction. This means, when the victim and spy have the same conditional branch, but different histories leading up to that branch, the TAGE predictor will calculate different indexes for the two branches, even though they have the same address. This is because the branch history is different between the victim and the spy, resulting in different index values.

This is why the attack involving TAGE predictor requires the attacker to train the predictor with a specific input sequence, so that the predictor's internal state is manipulated to predict the outcome of the victim's conditional branch in a predictable way.

Referring to the prototype code in Figure 6.1, the attack can be broken down into 3 distinct loops. The outermost loop, iterates over the secret data array, and for each element, performs a set of iterations to collect timing measurements for that specific byte.

The middle loop iterates `ATTACK_SAME_ROUNDS` number of times, with the aim of averaging out any noise in the timing measurements.

Finally, the innermost loop that trains the branch predictor for certain rounds (`TRAIN_TIMES`) on a conditional branch instruction in the `branch()` function. During the training phase, an array of all 0's or 1's is accessed using the *randIdx*, and at the last iteration *attackIdx* is used to access the secret data array and have the branch predictor predict the outcome of the branch.

The variable *attackIdx* is used to select the byte of `sec_data[]` that is targeted by the side-channel attack. The value of *attackIdx* is dependent on the memory layout of the target system, specifically the relative positions of the `sec_data[]` and `array1[]` arrays in memory.

In the given prototype code, the memory addresses of `sec_data[]` and `array1[]` are subtracted to obtain the initial value of attackIdx. This value represents the offset in bytes between the two arrays in memory. The reason for using this offset is that the attacker can

control the input value passed to the `branch()` function, and by selecting a specific index of array1 based on the value of attackIdx, the attacker can indirectly access the corresponding byte of `sec_data[]`.

By comparing the difference in cycles taken during each training round and the final attack round, the attacker can infer the secret data.

Let's say for example, the *attackIdx* points to a '0' in the secret data array. The branch predictor is trained on a set of 1's to execute the conditional branch instruction using the *randIdx*. After training, the *attackIdx* is passed as the input to `branch()` function and since the branch was previously trained on value '1', it will mispeculate for the *attackIdx* value, and so it would result in longer execution time compared to if the *attackIdx* pointed to '1' instead.

Compared to BRAD-v1, there is aren't individual victim and spy blocks because the `branch()` function itself acts as both the victim and the spy. The `branch()` function takes an input argument and executes either of the two code blocks based on the value of the argument. As explained before the `branch()` function serves as the victim because it is the function that is being attacked. It also serves as the spy because it is the function that is being used to gather the side-channel information.

6.2 A Slight Complication

As indicated in the previous section the *attackIdx* represents the offset in bytes between the two arrays (`sec_data[]` and `array1[]`) in memory. However, depending on the memory address of both these arrays, if the value of *arrayIdx* is large that means longer memory access latency, resulting in large difference in cycle count when accessing an element from

`array1[]` compared to accessing an element from `sec_data[]`.

This issue was very prominently seen when `array1[]` was defined with all zeros, which resulted in the array being stored at a different memory location compared to where the code was stored in memory. So, during the training phase when the randIdx would access `array1[]` it would result in greater execution cycles compared to when *attackIdx* accessing the `sec_data[]`.

To mitigate this issue, I tried using the `` `-O0' `` compiler flag that forces no code optimization when compiling, but this didn't help. So, instead an inexpensive solution was to add an extra '1' at the end of `array1[]` to make it store in the same memory space as the overall attack code.

6.3 Evaluation

6.3.1 Simulator

Similar to BRAD-v1, this attack also doesn't work on Verilator due to its inaccuracy in modeling the precise timing behavior. Verilator being cycle-accurate simulator, models the behavior of the processor at each clock cycle, including the execution of instructions, the updating of registers and memory, and the generation of signals and interrupts. The simulator tracks the state of the processor at each cycle and verifies that it meets the design specifications.

From a hardware point of view, the attack works with the timing of the branch instruction affected by the propagation delay of the signals in the design. These propagational delays must not be accurately modelled on Verilator resulting in no timing differences and therfore failure of replicating the side-channel attack.

6.3.2 FPGA

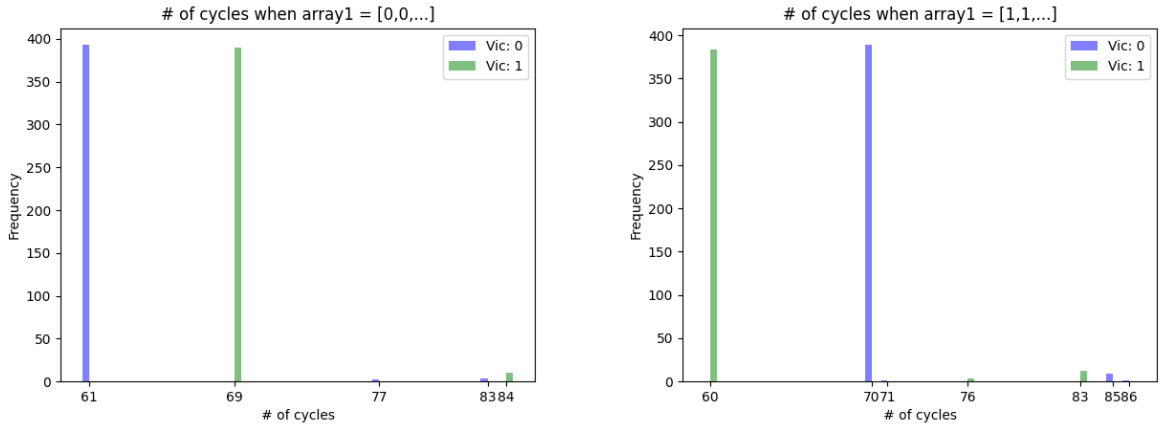
Since this attack was tailored to work with the TAGE branch predictor, I uploaded the new bitstream on to the FPGA with the TAGEBPD enabled.

To keep results from all attacks consistent, I have used the same 8-bit secret data array that has an even distribution of 1's and 0's arranged randomly.

To verify the working of the attack initially, I set `TRAIN_TIMES` to 6, and the 7th round would be the attack round with the *attackIdx*. I took 100 measurements for each set of 6 training rounds and 1 attack round. These measurements were stored in a text file and parsed to generate 7 plots in total. Out of those 7 plots, the first 6 are exactly similar to each other, as they are accessing the elements from the same array (`array1[]`).

The 7th graph is the one where *attackIdx* is used to access the secret data array (`sec_data[]`). Looking at the graphs in Figure 6.2, again there are 2 scenarios, one for the `array1[]` value with all 0's (Figure 6.2a) and another with 1's (Figure 6.2b). When the conditional branch instruction has been training on value '0' from the `array1[]` it is going to execute the conditional branch accordingly when in the 7th round that *attackIdx* is used. If the *attackIdx* matches with the training round value (*randIdx*) the execution would continue as it predicted correctly, but if the *attackIdx* is different then the branch predictor would have a mispredict and cause longer execution time.

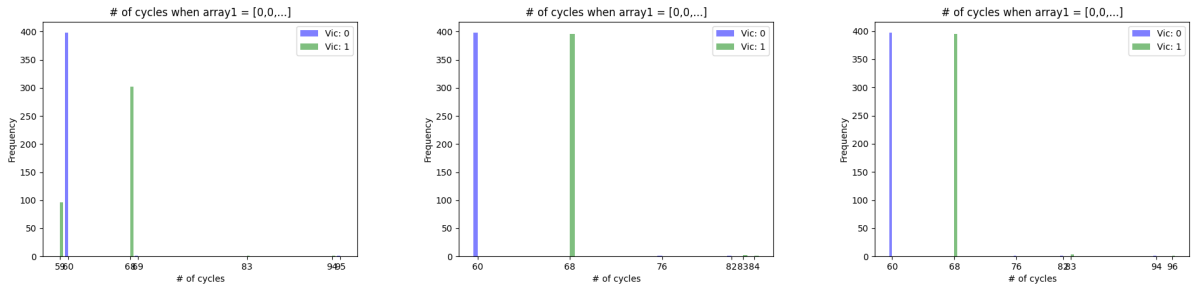
In section 6.1, I discussed two variables, `ATTACK_SAME_ROUNDS` and `TRAIN_TIMES`, which determine the number of attack rounds and training rounds, respectively, needed to access the secret array. The final step was to determine the minimum required `TRAIN_TIMES` before the attack index could be provided by the attacker to gain access to the secret array. Starting with just one training round, where the `branch()` function was called once on a random index and then using the attack index value, did not actually train the corresponding conditional



(a) `array1[]` is all 0's (b) `array1[]` is all 1's

Figure 6.2: Results for BRAD-v2 attack on FPGA

branch entry. However, when the number of training rounds was increased to 2, an observable change in the attack was observed. The advanced predictor, such as TAGEBPD, relies on the baseline predictor for its initial prediction, and since it corresponds to a 2-bit BIM table, it can be trained to be in the taken/not-taken state with just 2 rounds. Increasing the training rounds further did not affect the functionality of the attack, as demonstrated by the plots in Figure 6.3.



(a) `TRAIN_TIMES` = 1 (b) `TRAIN_TIMES` = 2 (c) `TRAIN_TIMES` = 3

Figure 6.3: Results for BRAD-v2 attack with multiple training rounds

Chapter 7

PINK Attack

Both the previous attacks were based on creating a BIM based side-channel attack using conditional branches. PINK (Pointer-based INdirect jump with Known targets) attack exploits BTB for the side-channel, but now based on unconditional indirect branches. The goal and overview for this attack is similar to what it was for BRAD-v1, only difference is that the BTB entry's target address is now dependant on an unconditional indirect jump instruction that redirects the execution path based on the correct/incorrect predictions.

Before diving into the details of the attack, it's essential to comprehend how program's branches work. Program's branches are instructions that allow a program to execute different paths depending on specific conditions. There are four classes of program branches: conditional direct, unconditional direct, conditional indirect, and unconditional indirect. Conditional direct branches are executed based on a certain condition and involve a jump to a specific instruction address. Unconditional direct branches are executed regardless of any conditions and also involve a jump to a specific instruction address. Conditional indirect branches are executed based on a condition, but the target address is determined at runtime by a value stored in a register or memory location. Unconditional indirect branches are executed without any conditions, but the target address is also determined at runtime by a value stored in a register or memory location. Out of the four types, only three - conditional direct, unconditional direct, and unconditional indirect - occur frequently [13]. The attack in question employs the use of unconditional indirect branches.

7.1 Detailed Design

Referring to the prototype code in Figure 7.1, both the victim and spy processes execute a code block that contains an indirect jump instruction, also known as a `goto` instruction.

```
uint8_t sec_data[8] = {1,0,0,1,...}; // Secret data
uint8_t array1[1] = {1}; // or 0

void indirJump(int val) {
    void* target_addr = (void*)(val*((uint64_t)&&T2) + (1-val)*((uint64_t)&&T1
    ));
    goto *target_addr;

    T1: __asm__ __volatile__("nop\n"
                "nop\n");
    T2: __asm__ __volatile__("nop\n"
                "nop\n");
}

int main(void) {
    for(int i=0; i<SEC_DATA_LEN; i++) {
        for(int k=0; k<ATTACK_ROUND; k++) {
            indirJump(sec_data[i]); // Victim call

            start = time();
            indirJump(array1[0]); // Spy call
            end = time();
            store_mispred_data(end - start);
        }
    }
}
```

Figure 7.1: Prototype code for PINK attack

In the `indirJump()` function, the target address of the jump is computed using the input 'val', which is used to select between two labels (*T1* and *T2*). The resulting address is then used as the target of the `goto` statement. Since the target address is computed dynamically at runtime based on the input value, this is an indirect jump.

Furthermore, since the jump is not conditioned on any previous instruction or register value,

it is an unconditional jump. The goto statement unconditionally transfers control to the target address computed by indirJump(), without any condition checks or branching. Therefore, this is an indirect unconditional jump.

When the victim accesses secret data, it uses the value of the data to determine which target to jump to, effectively training the processor's BTB to associate the value with the appropriate target.

When the spy runs, it also uses the same `goto` instruction to access the target addresses that may or may not have been previously accessed by the victim. Since the victim and spy use the same instruction, the BTB maps them to the same entry. The spy does not access the secret data itself but tries to infer its value based on the execution time of the instruction. Specifically, there are two possible settings under which the victim and spy blocks can execute:

- In the first setting, when the secret data is '1', both victim and spy execute the same target, **T2**, resulting in a BTB hit for the spy and a decrease in its access time.

- In the second setting, when the secret data is '0', the victim executes **T1** while the spy still attempts to execute **T2**, causing a BTB miss for the spy and longer access time.

By measuring the total execution time of the spy block under both settings, the secret data accessed by the victim can be deduced. Specifically, it can be inferred that if the spy block executes faster, the secret data value is '1', whereas if it executes slower, the secret data value is '0'.

7.2 TAGEBPD Limitations

The TAGE predictor implemented in BOOM is designed to handle the prediction of conditional branches and indirect jumps. However, it is not optimized for predicting the target addresses of unconditional jump instructions.

In BRAD-v2 attack, the attacker tried to exploit the predictor's misprediction behavior by training the predictor to predict the wrong outcome for a specific conditional branch instruction. On the contrary, PINK attack employs an indirect unconditional jump instruction that does not rely on previous instructions or register values, and the target address is computed dynamically at runtime based on input. Thus, the history of prior branch instructions cannot be used to predict the behavior of an indirect unconditional jump. As a result, the TAGEBPD does not need to be trained for attacks such as PINK that rely on an instruction that is not dependent on prior instruction or register values.

7.3 Evaluation and Verification

When evaluating the attack, I also thought of tweaking the original PINK attack code to change it to conditional indirect jump and compare the results with the primary PINK attack. Evaluation of both these versions of attack are summarized individually in their own sub-sections under Simulator and FPGA.

I made the following changes to the `indirJump()` function of the prototype code listed in Figure 7.2, so that the target address is actually depended on a conditional branch, but it is still an indirect jump to the respective target addresses (`T1` and `T2`).

```
...
void indirJump(int val) {
    void* target_addr = val ? (void*)&&T2 : (void*)&&T1;
    goto *target_addr;

    T1: __asm__ __volatile__("nop\n"
                "nop\n");
    T2: __asm__ __volatile__("nop\n"
                "nop\n");
}
...
```

Figure 7.2: Conditional Indirect Jump changes

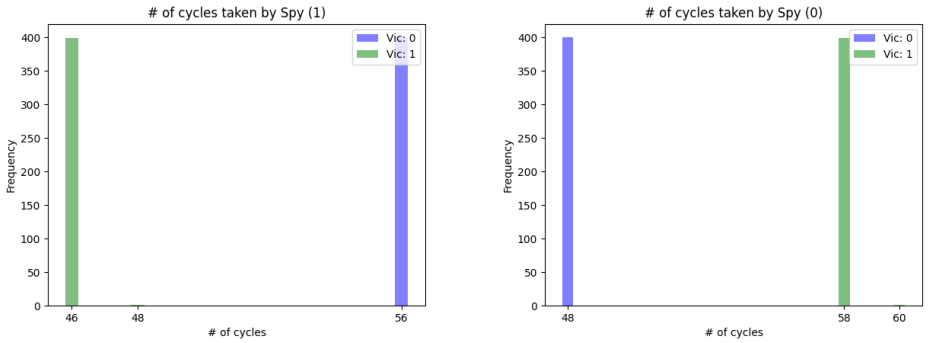
7.3.1 Simulator

Unconditional Indirect Jump

As mentioned in the earlier section, in the case of PINK attack, the branch predictor is not used to predict the target address of the jump, as the jump is always taken and its target address is determined at runtime based on the value of a register or memory location. The branch predictor does not know the target address of the jump until it is executed, so it cannot be trained to predict the target address. As a result, the attack works on Verilator because the branch predictor is not trained to handle this specific pattern of indirect jumps, and therefore the pattern is not detected as an anomaly. Figure 7.3 displays the simulation results I got after running the attack with Verilator.

I conducted the attack under two different scenarios. In the first scenario, the victim alternated between jumping to targets `T1` or `T2` based on a secret value, while the spy only jumped to T2 and recorded the number of cycles it took. In the second scenario, the spy jumped to `T1` and recorded the cycles. I repeated the attack with the spy jumping to `T1` to ensure consistency in the results. For each run, I kept about 100 attack rounds before saving the results onto a text file. It's worth noting that in Figure 7.1, because `T1` is located closer

to the jump instruction, the spy executed more instructions when jumping to `T1`. As a result, `T1's` measurements were higher than those of `T2` when the victim performed jumps to the matching target. However, the relative difference between the matching and mismatching targets was similar in both scenarios.



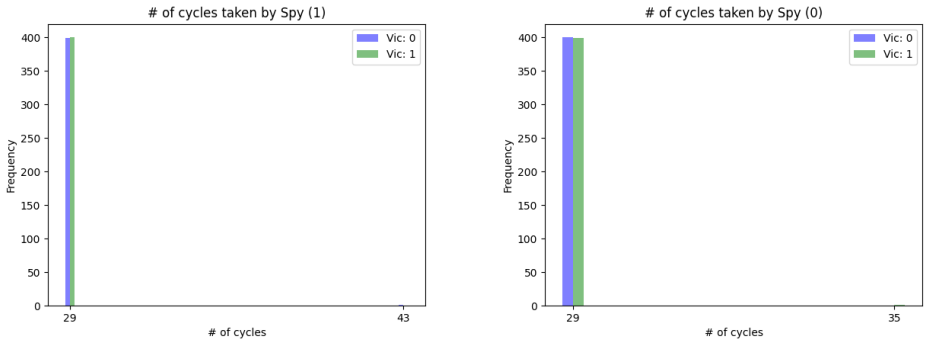
(a) Spy jumping to `T2` target (b) Spy jumping to `T1` target

Figure 7.3: Results for PINK attack on Verilator

Conditional Indirect Jump

In contrast, in the case of the conditional indirect jump attack, the branch predictor tries to predict the target address of the jump instruction, based on the past history of branch instructions. When the victim and spy access `T2` in an anomalous pattern, the branch predictor detects this pattern as anomalous and updates its prediction accordingly. This can lead to incorrect predictions of the target address, and the attack fails.

It is quite evident from Figure 7.4, that Verilator with conditional indirect jumps is no longer susceptible to the attack. The cycle count values for accessing the `T2` target by the victim are the same as when accessing the `T1` target, regardless of whether the spy is attempting to jump to the `T1` or `T2` target.



(a) Spy jumping to `T2` target (b) Spy jumping to `T1` target

Figure 7.4: Results for **conditional** indirect jump attack on Verilator

7.3.2 FPGA

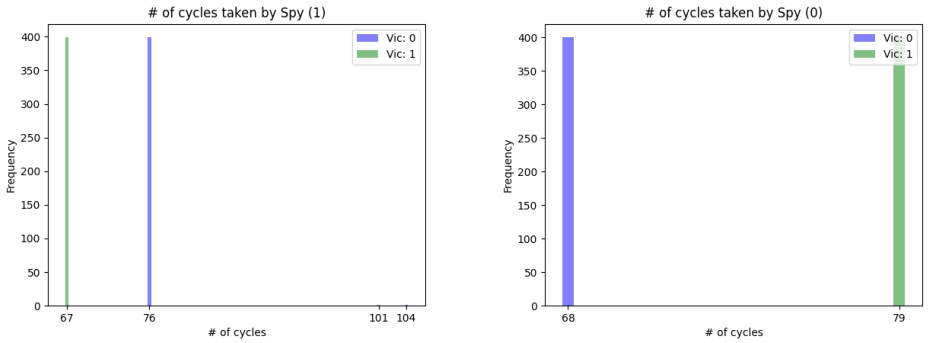
Unconditional Indirect Jump

Once I had enabled the TAGE backing predictor on the FPGA, I proceeded to replicate the steps I had taken when conducting the PINK attack on Verilator.

The results obtained are similar to those from the Verilator run. Although the total number of cycles has increased due to running the attack on physical hardware, the relative difference between the matching and mismatching targets remains consistent. These results are illustrated in Figure 7.5, and it is quite noticeable from the plots that when both the spy and victim execute the same target address the execution time is much lower compared to when they are different.

Conditional Indirect Jump

Switching from an unconditional indirect jump to a conditional indirect jump can improve the TAGEBPD's prediction accuracy, as the predictor can leverage the history of previous branch instructions to better predict the outcome of the conditional branch. However, since

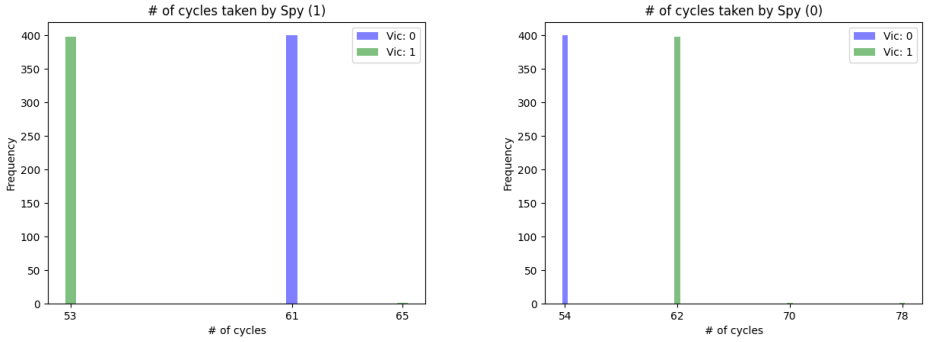


(a) Spy jumping to `T2` target (b) Spy jumping to `T1` target

Figure 7.5: Results for PINK attack on FPGA

it is still an indirect jump and the target address is not known beforehand, the attack can still be successful against the TAGEBPD predictor on the FPGA.

The TAGE predictor can predict the outcome of the conditional branch with more accuracy than the unconditional branch, which can result in lower execution time. However, the relative difference between matching and mismatching targets remains constant internally, as shown in Figure 7.6.



(a) Spy jumping to `T2` target (b) Spy jumping to `T1` target

Figure 7.6: Results for **conditional** indirect jump attack on FPGA

Chapter 8

PLoop Attack

The final attack that I demonstrated on the Small BOOM core was the PLoop (Predictive Loop) attack. In section 4.3.2, I talked a little bit about TAGE and Gshare branch predictors, but there is another type of predictor called the loop predictor that is often associated with TAGE and was first introduced by Andre Seznec [40].

8.1 Loop Predictor Outline

A loop predictor predicts the number of iterations a loop will execute for. It is used to improve performance for a loop-intensive program by allowing the core to execute instruction which are likely to be executed in future loop iterations.

The loop predictor in the TAGE predictor is used to predict the number of iterations of a loop by tracking the program counter (PC) of the loop header and the stride between consecutive iterations. The loop predictor maintains a table of loop headers, each with a corresponding entry that contains the stride and a counter that is incremented every time the loop header is encountered.

When the loop header is encountered, the loop predictor uses the PC to look up the corresponding entry in the loop table. If the PC is not found in the table, the loop predictor assumes that the loop is not a candidate for loop prediction and does not perform any pre-

dictions. Otherwise, the loop predictor uses the stride and the counter to predict the number of iterations of the loop.

If the loop predictor predicts that the loop will execute more than a threshold number of iterations, it prepares to executes the instructions in the loop body for the predicted number of iterations. If the loop actually executes fewer iterations than predicted, the execution is aborted and the processor continues with the correct execution path [39].

8.2 Attack Overview

In SonicBOOM, the TAGE-L branch predictor, which combines the TAGE and loop predictor, is enabled by default. I created a new configuration for only the loop predictor, in which it was enabled along with the BTB to store branches and their respective target addresses.

To perform the attack, I utilized the loop predictor in conjunction with secret data. Looking at the prototype code in Figure 8.1, I designed a function called `loop_f()` that included a for-loop whose iterations depended on a secret value. The loop function was designed to take an input value from either the victim or the spy, and based on that input value, it would run a loop a certain number of times.

The victim accessed the `loop_f()` function with their secret input, which determined the number of iterations of the loop. The secret input was either a '0' or a '1'. If the victim's secret input was 1, the loop would run for 3 iterations before exiting. If the victim's secret input was 0, the loop would run for 2 iterations before exiting. The spy, who did not know the secret value, also accessed the same `loop_f()` function with its own input value.

Now, the `loop_f()` function is called multiple times to train the loop predictor to predictively run the loop for certain number of times. This is specified by the `TRAIN_LOOP` number. This

causes the CPU to speculate the number of loop iterations for the data dependent for-loop inside the `loop_f()` function. By increasing this `TRAIN_LOOP` number I am essentially training the innermost loop in the `main()` function to provide a better prediction on the iteration for the data dependent loop.

After the innermost loop (`j-loop`) is trained in the `main()` function with victim's input, when the spy function is called with the same input as before, the loop predictor is going to predictively run the data dependent loop the same number of times as the victim did resulting in lower execution time. But if the values don't match, then the predictor would actually mispredict following a slower execution time.

It doesn't matter for how many iterations is the data dependent for-loop running for specific input values, what matters is the training rounds for that data dependent loop. For the predictor to predict the iterations it has to trained for certain rounds and this is explored more in section 8.3.2.

8.3 Evaluation and Verification

8.3.1 Simulator

PLoop attack also doesn't work on Verilator analogous to BRAD-v1 and BRAD-v2 attacks. For the same reason as mentioned before, the attacks are dependent on the difference in propagational delays and Verilator being a software simulator isn't able to accurately model that.

```
uint8_t sec_data[8] = {1,0,0,1,...}; // Secret data
uint8_t array1[1] = {1}; // or 0

void loop_f(uint8_t* addr) {
    for(int i=0; i<(*addr) + 2; i++) {
        __asm__ __volatile__("nop\n"
               "nop\n");
    }
}

void victim_f(uint8_t idx) {
    loop_f(&sec_data[idx]);
}
void spy_f(uint8_t idx) {
    loop_f(&array1[idx]);
}

int main(void) {
    for(int i=0; i<SEC_DATA_LEN; i++) {
        for(int k=0; k<ATTACK_ROUNDS; k++) {
            for(int j=0; j<TRAIN_LOOP; j++) {
                victim_f(sec_data[i]); // Victim call
            }
            start = time();
            spy_f(array1[0]); // Spy call
            end = time();
            store_loop_pred_data(end-start);
            }
        }
    }
}
```

Figure 8.1: Prototype code for PLoop attack

8.3.2 FPGA

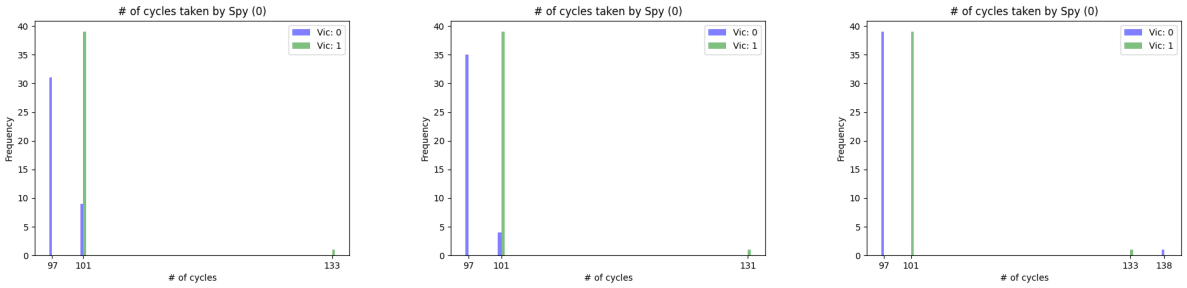
When trying to evaluate the efficacy of the attack, the primary thing was to get the least number of training rounds needed to execute the data dependent for-loop without any mis-predictions.

So, I started with initializing `TRAIN_LOOP` to 1 and gathered the data after running it on the FPGA. For sake of simplicity and understanding the output at first I kept the `ATTACK_ROUNDS`

to 10 and spy is accessing a value '0'. Looking at Figure 8.2a, the `j-loop` is only ran once, and even when the victim and spy both access '0', there are some mispredictions initially resulting in higher clock cycles (101) and once its trained after just 3 iterations it lowers down to 97 cycles. This 3 iterations are actually based on the confidence value of the loop predictor entry. At every iteration of the data dependent loop the confidence value is increased and once it reaches the threshold value the table entry can confidently predict the iterations of the for-loop.

As the training rounds increase, it results in reduced mispredictions, and this is clearly seen in Figure 8.2b. Even after the victim was trained for four rounds, the loop predictor still mispredicted the loop iteration, but it was much less than when the victim was trained only once.

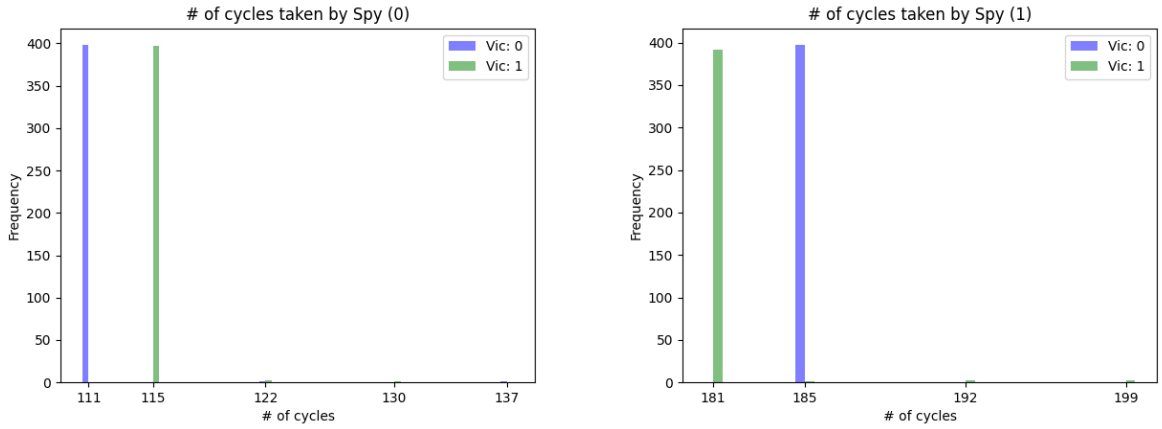
Moving forth, I continued to increase the value of `TRAIN_LOOP`, I found out that when it reached 7, there were no mispredictions when the spy timed its execution, as shown in Figure 8.2c. Since the victim and spy accessed the same value, the loop predictor was very accurate in predicting the number of iterations to run the loop. This value of 7 corresponds to the confidence value set in the loop predictor. The confidence value of a loop is updated every time the prediction of iterations matches the actual iteration of the loop. When the victim is trained seven times, the loop predictor updates this confidence value at each training round. When the spy accesses the same value, the loop predictor is confident enough to let the data-dependent for loop run the same number of iterations as it did for the victim.



(a) `TRAIN_LOOP` = 1 (b) `TRAIN_LOOP` = 4 (c) `TRAIN_LOOP` = 7

Figure 8.2: Results for PLoop attack with different training rounds

Moving on, I modified the data-dependent for-loop to perform a greater number of iterations when the data value is '1', aiming to achieve a more noticeable difference in cycle counts compared to when it is '0'. After knowing the minimum training rounds required, I ran the attack again for two different spy values i.e., '1' and '0' individually to notice the cycle differences between the two. I also took 100 measurements to reduce noise from the data and the results are depicted in Figure 8.3.



(a) Spy accessing value `0` (b) Spy accessing value `1`

Figure 8.3: Results for PLoop attack on FPGA

Chapter 9

Summary of Attacks

Each attack described in previous chapters was designed to target a specific branch predictor or a subset of predictors. Therefore, depending on the target predictor, different modules of the BOOM branch predictor can be enabled or disabled to customize the predictor.

The BOOM RISC-V branch predictor consists of five main modules - BTB, BIM, uBTB[1] (micro-BTB), Tage, and Loop, which can be used in various combinations to implement different predictors in the front-end of BOOM.

For instance, the NLP is the baseline predictor used in BOOM for its fast predictions and it includes the BTB, BIM, and uBTB modules. The TAGE-L is a state-of-the-art predictor that interacts with sub-predictors like NLP and Loop, and it comprises the BIM, uBTB, BTB, Tage, and Loop modules. The Loop predictor is usually used in combination with NLP and includes the BIM, uBTB, BTB, and Loop modules.

By selecting the appropriate modules and their combinations, various predictors such as NLP, TAGE-L, and Loop can be implemented as the main branch predictor in the BOOM front-end. Therefore, depending on the requirements of the attack, different predictors can be employed to optimize the performance of the BOOM RISC-V processor.

To make it easier to understand which attacks work on which set of modules, I have created a user-friendly table summarizing the information (Table 9.1). The table lists each attack

[1]The uBTB is a small buffer that predicts simple conditional branch outcomes but has limited capacity and cannot predict complex conditional branches.

and the set of modules required for the attack to be successful. The table serves as a helpful guide for selecting the appropriate modules and predictors for a given attack.

Table 9.1: Summary of the attacks on multiple set of Modules

BPU modules					**Side-channel attacks**			
BIM	uBTB	BTB	Loop	TAGE	BRAD-v1	BRAD-v2	PINK	PLoop
○	●	○	○	○	✓	✓	✓	✗
●	○	●	○	○	✓	✓	✓	✗
●	●	●	○	○	✓	✓	✓	✗
○	○	●	●	○	✗	✗	✓	✗
●	○	●	●	○	✓	✓	✓	✓
○	●	○	●	○	✓	✓	✓	✓
●	●	●	●	○	✓	✓	✓	✓
○	●	○	○	●	✗	✓	✓	✗
●	○	●	○	●	✗	✓	✓	✗
●	●	●	○	●	✗	✓	✓	✗
●	●	●	●	●	✗	✓	✓	✗

Referring to the attack columns in Table 9.1, the BRAD-v1 attack is effective when the NLP is enabled but fails to work when TAGE is enabled. However, both the BRAD-v1 and BRAD-v2 attacks rely on the BIM table, rendering them ineffective when this specific module is disabled. Conversely, the PINK attack demonstrates its versatility by being successful in all combinations of modules, as long as the BTB is enabled. Similarly, the PLoop attack specifically targets the loop predictor. Thus, if the loop predictor is disabled or lacks the necessary accompanying modules to provide the base prediction, the attack does not function.

Chapter 10

Mitigation Techniques

Mitigating the risks associated with side-channel attacks is crucial to maintain the security and privacy of computer systems. As already seen from the previous chapters, side-channel attacks allow an attacker to gain unauthorized access to confidential data without being detected. This makes it even harder to come up with mitigation techniques for these attacks, but failure to do so could result in severe consequences.

The mitigation techniques can be classified into software and hardware, and exploring these defenses for the attacks could be an interesting topic for future research.

10.1 Software Mitigation Techniques

Software-based countermeasures are restricted as they cannot control the mapping of individual branches inside a BPU, nor they can eliminate the source of leakage from the hardware itself. However, they can be effective sometimes and considering they are software-based they can be easily deployed on existing systems without requiring hardware modifications.

Return Trampoline (retpoline). Retpoline is a software-based mitigation technique designed to protect against the indirect branch target injection attack, a variant of the Spectre attack. This technique replaces vulnerable indirect branches with a special trampoline code that prevents CPUs from predicting the outcome of a branch that could indirectly leak sensi-

tive information. The trampoline code is constructed to cause the CPU to take a predictable path, preventing the leakage of sensitive information.

Retpoline works by exploiting the CPU's behavior when executing indirect branches. When an indirect branch is encountered, the CPU uses a branch predictor to predict the target of the branch. If the prediction is incorrect, the CPU must discard any instructions that were fetched and restart execution from the correct target. Retpoline prevents exploitation of this process by using a sequence of indirect jumps to return control to the calling function. The jumps are constructed in a way that always leads to the correct target, preventing the CPU from speculating on incorrect targets and leaking sensitive information through a side-channel [5]. While this mitigation technique is effective against the PINK attack, it does introduce additional overhead due to deliberate contamination of the RSB for swapping the indirect branch with return statements.

If-conversion. This is a compiler optimization technique that transforms a program's control flow graph by converting conditional code to sequential code. This mitigation technique converts control dependencies into data dependencies, thereby reducing the vulnerability of a program to timing side-channel attacks that rely on the direction of conditional branches [14, 20]. It is straightforward to apply this technique to simple branches with few dependencies, but converting complex control flow that executes different code based on branch outcomes is more challenging. The attacks that can be affected by this software countermeasure would BRAD-v1 and BRAD-v2, considering they depend on the direction of conditional branch in the code. Although this technique could be effective against attacks like BRAD-v1 and BRAD-v2, it should be noted that converting complex control flow to sequential flow remains challenging, leaving the possibility of vulnerabilities in certain cases.

10.2 Hardware Mitigation Techniques

First and the foremost technique to mitigate attacks lies in Table 9.1. By enabling certain advanced branch predictors like the TAGE-L backing predictor we can straight away mitigate BRAD-v1 attack which definitely worked with a basic branch predictor i.e., the NLP. In contrast to this, a naive approach to prevent side-channel attacks is to disable the BPU for sensitive branches or just reduce its accuracy so that its never used to speculate any instruction [20]. This could surely prevent most side-channel attacks, however at the cost of performance overhead.

Invisible BIM entry. To prevent side-channel attacks, a dedicated section of the Bi-Modal (BIM) table can be created for changed branches. Each entry in this section is associated with a unique process ID. Instead of directly updating the corresponding BIM table entry, the processor stores the updated BIM table state and the process ID separately when a branch is resolved. This approach prevents the outcome of the predictive branch from being leaked through the BIM table, making side-channel attacks more difficult. When the corresponding branch is committed, the BIM table entry is merged with the original entry to ensure consistency [26]. This technique is effective against all previously mentioned attacks. However, it does introduce performance and area overhead to the BIM since it needs to keep track of all branches whose predictions were updated before their commitment.

Virtual addresses. In order to avert BTB side-channel attacks, the BTB addressing mechanism can be changed to fully virtual address by modifying the indexing function. Instead of using the physical address of the branch target to index into the BTB, the virtual address can be used. This can be achieved by adding a new virtual tag array to the BTB, which stores the virtual address of the branch target along with other relevant information [19]. Let's say there are two processes running on a system, and each process has its own

virtual address space. Both processes have a conditional branch instruction that jumps to the same physical address, but the virtual addresses of their branch targets are different. This means that the two processes will end up indexing different entries in the BTB, even though they are jumping to the same physical address. As a result, attackers cannot use timing or power side-channel attacks to infer the virtual address's location within the BTB. It should be noted that implementing this technique can be costly due to the requirement of additional bits, as the tag size will considerably increase. Nonetheless, it serves as an effective countermeasure against the PINK attack.

Chapter 11

Real World Application

With the increasing popularity of RISC-V cores for their high scalability and customizable architecture, we are seeing an increasing number of embedded devices, such as IoT devices, adopting RISC-V cores. However, as with any embedded device, security is a major concern, and a potential vulnerability in one component of the system could compromise the entire system. After exploring some of the attacks in the earlier chapters they can be expanded to real-life use cases leaving many of these IoT devices compromised [7].

One such example vulnerable to side-channel attacks could be a smart lock system. Such systems typically use cryptographic algorithms to secure the lock and unlock process, and the cryptographic keys used in these algorithms are usually stored in a secure location, such as an encrypted memory region or a dedicated hardware module.

In a typical smart lock, the cryptographic keys are stored in EEPROM (Electrically Erasable Programmable Read-Only Memory) along with the micro-controller inside the lock. When a user enters a 6-digit code, the smart lock starts to compare the code entered by the user to the code stored in EEPROM, one digit at a time. If the codes don't match, the operation is aborted, and the lock buzzes [25]. This is where a timing side-channel attack comes into the picture.

Consider the following pseudocode in Figure 11.1. The attack involves timing the for-loop that checks the correctness of each digit. If the entered digit checks out with the true digit,

each iteration takes about 100 cycles. However, if the entered digit is wrong, the loop ends, and the lock buzzes out. By timing the for-loop and having it go over all the 10 (0 through 9) possibilities for each digit, the longest time delay encountered for a specific digit tells us about the correctness of that digit. This process can be repeated for the remaining digits to infer the actual code.

```
bool verify_keycode(int enter_code[6], int true_code[6]) {
    for (int i=0; i<6; i++) {
        if enter_code[i] != true_code[i]
            return false;
    }
    return true;
}
```

Figure 11.1: Pseudocode for smart-lock keycode comparison

Usually, this type of timing related vulnerability in the code is often compensated by having an 'else' condition along-with the if. The 'else' condition is populated with additional instructions that match in timing with the 'if' condition, so that when an attacker times the for-loop's execution for each iteration, it will still execute to the end even if one the bits entered were wrong. The attacker now wouldn't be able to infer which exact branch condition was executed and thus brute forcing sequences wouldn't work anymore. The pseudocode for this fix in the original code is mentioned in Figure 11.2.

Even though, I took out the timing dependency by modifying the code so that it doesn't terminate midway if an incorrect digit is encountered, it is still susceptible to attack similar to BRAD-v1. The attacker can still infer the direction of the branch as the conditional statement is still there. Consider the following attack scenario, the attacker fixes the first 5 digits and tries all the 10 different combinations for the last digit. In the training phase the attacker calls the `verify_keycode()` function (to train the BIM entry to either Taken/Not-

```
bool verify_keycode(int enter_code[6], int true_code[6]) {
    int correct = 1;
    for (int i=0; i<6; i++) {
        if enter_code[i] != true_code[i]
            correct = 0;
        else
            correct = correct && 1;
            asm("nop") // additional inst to balance the timing
    }
    return (correct==1);
}
```

Figure 11.2: Alteration in pseudocode for smart-lock

taken state) for each of those 10 possibilities. After each training phase for a specific last digit, the attacker observes its execution again for a specific 6-digit code, where the first 5 digits are same as before and the last digit is a new one.

Let's consider an example where the desired final digit is '2'. In the training phase, the attacker begins with a last-digit value of 'xxxxx0' and increments it in each training round. The attacker measures the execution time for each round. It is assumed that the conditional branch is taken for the correct digit and not taken for an incorrect one. Therefore, when the attacker trains the branch with the correct digit 'xxxxx2', the branch will be taken. However, when the attacker runs the fourth combination and measures the time, a misprediction occurs, resulting in a longer execution time for 'xxxxx3' compared to other training rounds. This enables the attacker to deduce the true key code digit-by-digit, allowing them to exploit the vulnerability of the branch predictor using a timing side-channel attack.

Chapter 12

Future Work

The successful implementation of side-channel attacks like BRAD-v1, BRAD-v2, PINK, and PLoop in the front-end of the BOOM core highlight the potential vulnerabilities in this area. However, these attacks only target a subset of the components in the front-end, leaving other components unexplored.

One such component is the RSB, which can be leveraged to extract sensitive information through side-channel attacks. The RSB plays a critical role in the control flow of the processor, making it a valuable target for attackers seeking to manipulate program execution or gain access to confidential information.

Given the potential vulnerabilities in the RSB, it is necessary to investigate the feasibility of side-channel attacks targeting this component and develop effective mitigation techniques to prevent such attacks. This would involve designing and implementing test cases to simulate potential attacks and evaluating the effectiveness of proposed countermeasures.

To evaluate the effectiveness of the proposed mitigation techniques, further research is needed to implement them and test their ability to prevent side-channel attacks targeting the front-end of BOOM. By identifying and addressing potential vulnerabilities in the RSB and other components of the front-end, it is possible to enhance the security of the BOOM processor and protect against a range of security threats.

Chapter 13

Conclusions

Side-channel attacks pose a significant threat to the security of modern processors. While the BOOM core is gaining popularity due to its customizable and scalable architecture, it is vulnerable to these attacks. The attacks presented definitely demonstrate the vulnerability of the BOOM core's front-end to side-channel attacks, highlighting the importance of identifying and addressing vulnerabilities in processor designs. By doing so, researchers can help ensure the security of various applications where the BOOM core is used, including IoT, embedded systems and AI/ML accelerators where security is of utmost importance. Further research into the vulnerabilities of different processor architectures is necessary to develop effective countermeasures to mitigate these threats and improve the overall security of modern processors.

Bibliography

[1] Welcome to RISCV-BOOM's documentation! — RISCV-BOOM documentation. URL https://docs.boom-core.org/en/latest/index.html.

[2] Welcome to chipyard's documentation (version "1.9.0")! — chipyard 1.9.0 documentation. URL https://chipyard.readthedocs.io/en/stable/index.html.

[3] Indirect branch predictor barrier, 2018. URL https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-predictor-barrier.html.

[4] Indirect branch restricted speculation, 2018. URL https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-restricted-speculation.html.

[5] Retpoline: A Branch Target Injection Mitigation, 2022. URL https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/retpoline-branch-target-injection-mitigation.html.

[6] Mohammad Ali Nassiri Abrishamchi, Abdul Hanan Abdullah, Adrian David Cheok, and Kevin S. Bielawski. Side channel attacks on smart home systems: A short overview. In *IECON 2017 - 43rd Annual Conference of the IEEE Industrial Electronics Society*, pages 8144–8149. IEEE. ISBN 978-1-5386-1127-2. doi: 10.1109/IECON.2017.8217429. URL http://ieeexplore.ieee.org/document/8217429/.

[7] Mohammad Ali Nassiri Abrishamchi, Abdul Hanan Abdullah, Adrian David Cheok, and

Kevin S. Bielawski. Side channel attacks on smart home systems: A short overview. In *IECON 2017 - 43rd Annual Conference of the IEEE Industrial Electronics Society*, pages 8144–8149, 2017. doi: 10.1109/IECON.2017.8217429.

[8] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib Ul Hassan, Cesar Pereida Garcia, and Nicola Tuveri. Port contention for fun and profit. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 870–887. IEEE. ISBN 978-1-5386-6660-9. doi: 10.1109/SP.2019.00066. URL https://ieeexplore.ieee.org/document/8835264/.

[9] Krste Asanovic, Rimas Avižienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Palmer Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Benjamin Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The rocket chip generator.

[10] Krste Asanovic, David A Patterson, and Christopher Celio. The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor. Technical report, University of California at Berkeley Berkeley United States, 2015.

[11] Ruxandra Bălucea and Paul Irofti. Software mitigation of risc-v spectre attacks. *arXiv preprint arXiv:2206.04507*, 2022.

[12] 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. ISBN 978-1-939133-06-9. URL https://www.usenix.org/conference/usenixsecurity19/presentation/canella.

[13] Po-Yung Chang, Eric Hao, and Yale N Patt. Target prediction for indirect jumps. *ACM SIGARCH Computer Architecture News*, 25(2):274–283, 1997.

[14] Youngsoo Choi, Allan Knies, Luke Gerke, and Tin-Fook Ngai. The impact of if-conversion and branch prediction on program execution on the intel itanium processor. pages 182–191. doi: 10.1109/MICRO.2001.991117.

[15] Md Hafizul Islam Chowdhuryy and Fan Yao. Leaking secrets through modern branch predictor in the speculative world. pages 1–1. ISSN 0018-9340, 1557-9956, 2326-3814. doi: 10.1109/TC.2021.3122830. URL http://arxiv.org/abs/2107.09833.

[16] Elke De Mulder, Samatha Gummalla, and Michael Hutter. Protecting RISC-v against side-channel attacks. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–4. ACM. ISBN 978-1-4503-6725-7. doi: 10.1145/3316781.3323485. URL https://dl.acm.org/doi/10.1145/3316781.3323485.

[17] Shuwen Deng, Bowen Huang, and Jakub Szefer. Leaky frontends: Security vulnerabilities in processor frontends. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 53–66. IEEE, 2022.

[18] Alexander Dörflinger, Mark Albers, Benedikt Kleinbeck, Yejun Guan, Harald Michalik, Raphael Klink, Christopher Blochwitz, Anouar Nechi, and Mladen Berekovic. A comparative survey of open-source application-class RISC-v processor implementations. In *Proceedings of the 18th ACM International Conference on Computing Frontiers*, pages 12–20. ACM. ISBN 978-1-4503-8404-9. doi: 10.1145/3457388.3458657. URL https://dl.acm.org/doi/10.1145/3457388.3458657.

[19] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *2016 49th Annual IEEE/ACM Inter-*

national Symposium on Microarchitecture (MICRO), pages 1–13. IEEE, . ISBN 978-1-5090-3508-3. doi: 10.1109/MICRO.2016.7783743. URL http://ieeexplore.ieee.org/document/7783743/.

[20] Dmitry Evtyushkin, Ryan Riley, Nael Cse {And} Ece Abu-Ghazaleh, and Dmitry Ponomarev. BranchScope: A new side-channel attack on directional branch predictor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 693–707. ACM, . ISBN 978-1-4503-4911-6. doi: 10.1145/3173162.3173204. URL https://dl.acm.org/doi/10.1145/3173162.3173204.

[21] Neel Gala, Arjun Menon, Rahul Bodduna, G. S. Madhusudan, and V. Kamakoti. SHAKTI processors: An open-source hardware initiative. In *2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID)*, pages 7–8. IEEE. ISBN 978-1-4673-8700-2. doi: 10.1109/VLSID.2016.130. URL http://ieeexplore.ieee.org/document/7434907/.

[22] Michael Gielda. GitHub - riscvarchive/riscv-cores-list: RISC-V Cores, SoC platforms and SoCs, 2021. URL https://github.com/riscvarchive/riscv-cores-list.

[23] Abraham Gonzalez, Ben Korpan, Jerry Zhao, Ed Younis, and Krste Asanović. Replicating and mitigating spectre attacks on an open source RISC-V microarchitecture. In *Third Workshop on Computer Architecture Research with RISC-V (CARRV 2019)*, Phoenix, AZ, USA, 2019.

[24] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+flush: A fast and stealthy cache attack. In Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 9721, pages 279–299. Springer International Publishing. ISBN 978-

3-319-40666-4 978-3-319-40667-1. doi: 10.1007/978-3-319-40667-1_14. URL http://link.springer.com/10.1007/978-3-319-40667-1_14. Series Title: Lecture Notes in Computer Science.

[25] Plore Hacker. Side channel attacks on high security electronic safe locks. doi: 10.5446/36280. URL https://av.tib.eu/media/36280. Publisher: DEF CON.

[26] Md Hafizul Islam Chowdhuryy, Hang Liu, and Fan Yao. BranchSpec: Information leakage attacks exploiting speculative branch instruction executions. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pages 529–536. IEEE. ISBN 978-1-72819-710-4. doi: 10.1109/ICCD50377.2020.00095. URL https://ieeexplore.ieee.org/document/9283585/.

[27] Mustakimur Khandaker, Abu Naser, Wenqing Liu, Zhi Wang, Yajin Zhou, and Yueqiang Cheng. Adaptive call-site sensitive control flow integrity. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 95–110. IEEE. ISBN 978-1-72811-148-3. doi: 10.1109/EuroSP.2019.00017. URL https://ieeexplore.ieee.org/document/8806734/.

[28] 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 *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE. ISBN 978-1-5386-6660-9. doi: 10.1109/SP.2019.00002. URL https://ieeexplore.ieee.org/document/8835233/.

[29] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael B Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *WOOT@ USENIX Security Symposium*, 2018.

[30] Norbert Kremeris. Verilator pt.1: Introduction. URL https://itsembedded.com/dhd/verilator_1/.

[31] 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)*, pages 973–990, Baltimore, MD, August 2018. USENIX Association. ISBN 978-1-939133-04-5. URL https://www.usenix.org/conference/usenixsecurity18/presentation/lipp.

[32] Fangfei Liu and Ruby B. Lee. Random fill cache architecture. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 203–215. IEEE. ISBN 978-1-4799-6998-2. doi: 10.1109/MICRO.2014.28. URL http://ieeexplore.ieee.org/document/7011389/.

[33] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622. IEEE. ISBN 978-1-4673-6949-7. doi: 10.1109/SP.2015.43. URL https://ieeexplore.ieee.org/document/7163050/.

[34] Robert Martin, John Demme, and Simha Sethumadhavan. TimeWarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 118–129. IEEE. ISBN 978-1-4673-0476-4 978-1-4673-0475-7 978-1-4673-0473-3 978-1-4673-0474-0. doi: 10.1109/ISCA.2012.6237011. URL http://ieeexplore.ieee.org/document/6237011/.

[35] David A. Patterson and John L. Hennessy. *Computer Organization & Design: The*

Hardware/Software Interface. Morgan Kaufmann Publishers, San Francisco, California, 1994. ISBN 1558602828.

[36] Ivan Puddu, Moritz Schneider, Miro Haller, and Srdjan Capkun. Frontal attack: Leaking control-flow in sgx via the cpu frontend. In *USENIX Security Symposium*, pages 663–680, 2021.

[37] 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. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 361–374. IEEE. ISBN 978-1-66543-333-4. doi: 10.1109/ISCA52012.2021.00036. URL https://ieeexplore.ieee.org/document/9499837/.

[38] André Seznec. A 64-kbytes ittage indirect branch predictor. In *JWAC-2: Championship Branch Prediction*, 2011.

[39] André Seznec. A new case for the TAGE branch predictor. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 117–127. ACM. ISBN 978-1-4503-1053-6. doi: 10.1145/2155620.2155635. URL https://dl.acm.org/doi/10.1145/2155620.2155635.

[40] André Seznec. Tage-SC-L branch predictors. In *Proceedings of the 4th Championship on Branch Prediction*, 2014. URL http://www.jilp.org/cbp2014/.

[41] Suse Support. Security vulnerability: "meltdown" and "spectre" side channel attacks against CPUs with speculative execution. (7022512). URL https://www.suse.com/support/kb/doc/?id=000019105.

[42] Jakub Szefer. Principles of secure processor architecture design. *Synthesis Lectures on Computer Architecture*, 13(3):1–173, 2018.

[43] Jakub Szefer. Survey of microarchitectural side and covert channels, attacks, and defenses. *Journal of Hardware and Systems Security*, 3(3):219–234, 2019.

[44] Rodothea Myrsini Tsoupidi, Elena Troubitsyna, and Panagiotis Papadimitratos. Thwarting code-reuse and side-channel attacks in embedded systems. *arXiv preprint arXiv:2304.13458*, 2023.

[45] 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. ISBN 978-1-72811-444-6. doi: 10.1109/HPCA.2019.00058. URL https://ieeexplore.ieee.org/document/8675222/.

[46] 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*, pages 572–586. ACM. ISBN 978-1-4503-6938-1. doi: 10.1145/3352460.3358306. URL https://dl.acm.org/doi/10.1145/3352460.3358306.

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

[48] Yuval Yarom and Naomi Benger. Recovering openssl ecdsa nonces using the FLUSH+RELOAD cache side-channel attack. *IACR Cryptology ePrint Archive*, 2014, 2014.

[49] Yuval Yarom and Katrina Falkner. Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 719–732, 2014.

[50] Florian Zaruba and Luca Benini. The cost of application-class processing: Energy

and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27 (11):2629–2640, 2019.

[51] Jerry Zhao, Abraham Gonzalez, Alon Amid, Sagar Karandikar, and Krste Asanovic. COBRA: A framework for evaluating compositions of hardware branch predictors. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 310–320. IEEE. ISBN 978-1-72818-643-6. doi: 10.1109/ISPASS51385.2021.00053.

[52] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. Sonicboom: The 3rd generation berkeley out-of-order machine. In *Fourth Workshop on Computer Architecture Research with RISC-V*, volume 5, 2020.