

SpecBMC

Bounded Model Checker for Speculative Non-Interference

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering/Internet Computing

eingereicht von

Emmanuel Pescosta, BSc

Matrikelnummer 01326934


an der Fakultät für Informatik
der Technischen Universität Wien


Betreuung: Ass. Prof. Dipl.-Ing. D.Phil. Georg Weissenbacher

Mitwirkung: Ass. Prof. Dipl.-Math. Dr.techn. Florian Zuleger

RA Dipl.-Ing. Thomas Pani

Wien, 10. Oktober 2020


Emmanuel Pescosta


Georg Weissenbacher



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

SpecBMC

Bounded Model Checker for Speculative Non-Interference

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering/Internet Computing

by

Emmanuel Pescosta, BSc

Registration Number 01326934

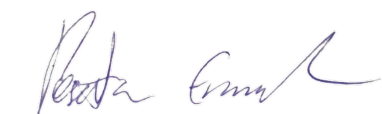
to the Faculty of Informatics
at the TU Wien


Advisor: Ass. Prof. Dipl.-Ing. D.Phil. Georg Weissenbacher

Assistance: Ass. Prof. Dipl.-Math. Dr.techn. Florian Zuleger

RA Dipl.-Ing. Thomas Pani

Vienna, 10th October, 2020


Emmanuel Pescosta


Georg Weissenbacher



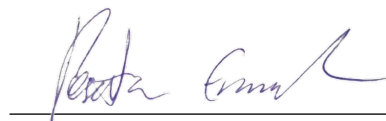
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Erklärung zur Verfassung der Arbeit

Emmanuel Pescosta, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 10. Oktober 2020



Emmanuel Pescosta



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Danksagung

Zuerst möchte ich mich bei meinen beiden Betreuern, Prof. Georg Weissenbacher und Prof. Florian Zuleger, für die tolle Betreuung und wertvollen Rückmeldungen während der gesamten Projektphase ausdrücklich bedanken. Die vielen interessanten Diskussionen und herausfordernden Fragen haben mich stets motiviert und mir immer sehr weitergeholfen. Bei Georg möchte ich mich speziell dafür bedanken, dass er mir diese interessante und lehrreiche Arbeit erst ermöglicht hat, dazu zähle ich auch seine großartige Vorlesung “Software Model Checking” ohne die ich diese Arbeit wohl nicht durchführen hätte können. Bei Florian möchte ich mich speziell dafür bedanken, dass er sich bereit erklärt hat mein zweiter Betreuer zu sein und für seine wertvollen Ideen und Vorschläge, durch diese ich meine Arbeit stets verbessern und vereinfachen konnte. Zudem möchte ich mich bei Thomas Pani für seine Ideen und Anregungen speziell in der Anfangsphase meiner Arbeit bedanken.

Zu tiefst möchte ich mich bei meinen Eltern, Martina und Oskar, für die liebevolle Unterstützung und Motivation während meines gesamten Bildungsweges bedanken. Zu guter Letzt möchte ich mich bei meiner Freundin Patricia herzlichst bedanken, die mir über all diese Zeit ihre Liebe und großartige Unterstützung geschenkt hat – ohne dich wäre diese Arbeit nicht möglich gewesen.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Kurzfassung

Moderne Prozessoren verwenden eine Vielzahl von Techniken für die Steigerung der Ausführungsgeschwindigkeit, wie etwa Sprungvorhersage, spekulative Ausführung und Instruktionsevaluierung unabhängig von der eigentlichen Programmreihenfolge, um nur ein paar davon zu nennen. All diese Techniken kombiniert können Geschwindigkeitsvorteile bringen, nämlich indem der Prozessor anstatt zu warten die nachfolgenden Instruktionen spekulativ ausführt. Bei Verzweigungen zum Beispiel, kann die mögliche Verzweigungsbedingung spekulativ angenommen werden anstatt zu warten bis die Bedingung tatsächlich ausgewertet werden kann. Ist die Annahme korrekt, so kann der Prozessor die Wartezeit mittels Vorberechnung vermeiden. Ist die Annahme jedoch inkorrekt, so werden die vorab berechneten Ergebnisse einfach verworfen und somit ist nur die Wartezeit “verschwendet”.

Jedoch haben Wissenschaftler im Jahr 2018 gezeigt, dass manche Seiteneffekte der spekulativen Ausführung auch nach dem Verwerfen der inkorrekten Spekulation erhalten bleiben. Diese übriggebliebenen Seiteneffekte erlauben es sensitive Informationen aus der spekulativen Ausführung heraus zu transferieren. Die sogenannten *Spectre* Attacken waren geboren [7, 28]. Obwohl *Spectre* zu der Klasse der Hardware-Fehler zählt, gibt es mehrere mögliche software-basierte Gegenmaßnahmen [7, 9, 25, 35, 41]. So kann zum Beispiel durch das sorgfältige Einfügen von Spekulationsbarrieren die spekulative Ausführung von verwundbaren Instruktionen unterbunden werden. Wie jedoch die Vergangenheit gezeigt hat, sind *Spectre* Schwachstellen schwierig zu finden und zu unterbinden, sowohl für Menschen als auch für Programm-Übersetzer.

Im Rahmen dieser Arbeit wird eine formale Semantikbeschreibung für spekulatives Ausführungsverhalten entwickelt. Diese Semantik erlaubt es uns sowohl Spectre-PHT als auch Spectre-STL Schwachstellen formal zu erfassen, die auf Kontrollfluss- beziehungsweise Datenfluss-Misspekulation beruhen. Des weiteren definieren wir eine Eigenschaft von Programmen die es uns erlaubt, Aussagen über die Sicherheit im Bezug auf *Spectre* Attacken zu treffen. Ein Programm wird als sicher angesehen, falls das Programm ausgeführt auf einem Prozessor mit spekulativem Verhalten die selben erkennbaren sicherheitssensitiven Seiteneffekte erzeugt, wie das identische Programm ausgeführt auf einem Prozessor ohne jegliche Spekulation. Basierend auf der formalen Beschreibung wird ein statisches Analysewerkzeug entwickelt um *Spectre* Schwachstellen effizient auffinden zu können. Das Analysewerkzeug, genannt SpecBMC, verwendet dazu das Konzept von Bounded Model-Checking. SpecBMC wird gegen eine Vielzahl von Spectre-PHT und

Spectre-STL Beispielprogrammen evaluiert, zudem zeigen wir das viele der vorgeschlagenen Gegenmaßnahmen nachweislich funktionieren. Zu guter Letzt führen wir eine kleine Fallstudie durch in der wir SpecBMC auf den Linux Kernel anwenden, dabei zeigen wir eine subtile Spectre-PHT Schwachstelle im Linux Kernel auf.

Das Resultat dieser Arbeit ist eine erweiterbare Semantikbeschreibung für spekulatives Ausführungsverhalten und ein statisches Programmanalyse-Werkzeug, das sowohl Spectre-PHT als auch Spectre-STL Schwachstellen in ausführbaren Programmen aufspüren kann.

Abstract

Modern processors employ a diverse set of techniques to improve their execution performance, such as branch prediction, speculative and out-of-order execution to name only a few of them. All these techniques combined allow to improve the computing performance, e.g., by predicting branch outcomes and transiently executing subsequent instructions if the processor would otherwise have to wait until the branch condition can finally be evaluated. If the prediction is correct the processor avoids a costly delay by pre-computing the results, while in case of a mis-prediction the speculatively computed results are simply thrown away and thus only the waiting time is “wasted”.

In the year 2018 researchers have shown, that some side effects of the mis-predicted speculative execution remain even after rollback, thereby allowing an adversary to leak sensitive information from within the speculative execution. The so-called *Spectre* attacks were born [7, 28]. Although *Spectre* belongs to the class of hardware bugs, some software-based countermeasures have been proposed [7, 9, 25, 35, 41]. For example, by carefully placing speculation barriers in the program code, speculative execution of potentially vulnerable instructions can be prevented. But as history has shown, *Spectre*-style vulnerabilities are hard to detect and mitigate, both for humans and compilers.

In the scope of this thesis a formal semantics for speculative execution is developed. The semantics allows us to formally capture Spectre-PHT and Spectre-STL vulnerabilities, which rely on control- respectively data-flow mis-predictions. Furthermore, we define a hyperproperty to reason about the security of a program in respect to speculative execution attacks. A program is considered secure, if the program executed on a CPU with speculation has the same adversary observable security sensitive side effects as if the program is executed on a CPU without speculation. Based on the formal definition, a bounded software model-checker for efficiently detecting Spectre-style vulnerabilities in binary programs is implemented. The tool is named SpecBMC. We validate SpecBMC against different Spectre-PHT and Spectre-STL examples and check if the different proposed countermeasures actually work. Finally, we conduct a small case study and show that SpecBMC is able to find a subtle Spectre-PHT bug in the Linux kernel.

The result of this thesis is an extensible speculative semantics and a static binary analysis tool that detects Spectre-PHT and Spectre-STL vulnerabilities in binary programs.

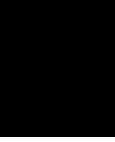


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

Kurzfassung	ix
Abstract	xi
1 Introduction	1
1.1 Motivation	2
1.2 Problem Statement	4
1.3 Methodological Approach	5
1.4 Notation	6
2 Intermediate Representation	9
2.1 Syntax	10
2.2 Default Semantics	11
3 Speculative Execution	13
3.1 Transient Execution Attacks	14
3.2 Setting	22
3.3 Transient Execution Semantics	22
3.4 Observations	26
3.5 Speculative Non-Interference	28
4 Microarchitecture	29
4.1 Cache	30
4.2 AVX Unit	35
4.3 Branch Predictor	38
4.4 Combination of Microarchitectural Components	43
5 Implementation	45
5.1 Important Concepts	46
5.2 Loader	47
5.3 SpecBMC	49
5.4 Solver	71
5.5 Additional Features	73

6	Evaluation	75
6.1	Spectre-STL	75
6.2	Spectre-PHT	78
6.3	Kocher Examples	82
6.4	Case Study	96
7	Related Work	103
7.1	Spectector	103
7.2	A Formal Approach to Secure Speculation	106
7.3	SCADET	107
7.4	Spectre is here to stay	108
7.5	CacheAudit	108
7.6	SpecFuzz	108
8	Conclusion	111
8.1	Future Work	112
A	Proofs	113
A.1	Trace obtained by $ns(\pi)$ is valid	113
A.2	Well-definedness of instruction's microarchitectural effects	114
A.3	Single speculative state is enough	115
B	Miscellaneous	127
B.1	Predictor Comparison	127
B.2	SpecBMC Environment Reference	132
	List of Figures	135
	List of Tables	137
	Bibliography	139



Introduction

Modern processors employ a diverse set of techniques to improve their execution performance [47], such as branch prediction, speculative and out-of-order execution or caches to name only a few of them. Branch prediction allows the processor to make educated guesses about which execution path will be taken most-likely when hitting a branching instruction. Out-of-order execution allows the processor to re-order instructions to reduce costly delays during execution, meaning that succeeding instructions may be executed as soon as their operands are available instead of waiting until all preceding instructions are finished. Caches reduce the average memory access latency [47] by buffering recently accessed memory locations in a small but fast cache memory.

Speculative execution in combination with out-of-order execution allows the processor to execute succeeding instructions in advance, even before the results of all preceding instructions are known [47]. Thereby, the processor tries to predict the most likely outcome of a yet unknown value and speculatively executes succeeding instructions using the predicted value. Once the actual value finally becomes available, the prediction is verified. If the prediction was correct, the speculatively computed results are applied and the execution continues. If the prediction was wrong, the speculatively computed results are discarded and the instructions are re-executed but this time using the actual value. The second case, a mis-prediction followed by a rollback, is usually denoted as *transient execution* [7]. In brief summary this means: speculative execution can increase the computing performance, namely if the prediction is correct the processor avoids a costly delay, while in case of a mis-prediction only the waiting time is wasted.

But then came *Spectre* [28], a whole new family of attacks exploiting hardware bugs in modern processors. In 2018 researchers have shown, that some effects of the transient execution remain even after rollback [28]. For example, after rollback the cache may still contain memory locations accessed during transient execution. While these remaining side effects don't negatively affect the correctness of the computation, they can reveal sensitive information about data accessed during transient execution. Since then a huge amount of Spectre attacks have been demonstrated and numerous software-

and hardware-based countermeasures have been proposed [7]. In this work we focus on software-based countermeasures and check their correct deployment.

1.1 Motivation

As the root cause of all Spectre-type attacks is the speculative behavior of modern CPUs, a simple yet efficient approach to prevent them would be to completely disable the speculative execution [28]. But this approach has serious negative consequences on the overall execution performance of modern CPUs [7], therefore finer-grained mitigation approaches are required. For example, speculation barriers allow to selectively prevent speculative execution where the program code is vulnerable to Spectre attacks [25], while for the rest of the program the CPU can still make use of speculative execution. The goal is to add enough mitigations such that all transient execution attacks are prevented, while avoiding unnecessary mitigations to minimize the possible performance losses. But as the following two examples show, this goal is hard to achieve both for humans and automated tools.

1.1.1 Linux Kernel: Backporting Error Reintroduced Spectre Vulnerability

In June 2019, Dianzhang Chen found a potential Spectre-PHT vulnerability in the Linux kernel. Via the `sys_ptrace()` system call an adversary was able to reach an unprotected array access in the `ptrace_get_debugreg()` function, shown in Listing 1.1. The input parameter `n`, which is used as the array index at line 7, is controllable from userspace. An adversary could cause a speculative out-of-bounds read and thereby load secret information into `bp` during transient execution. By using `bp` as a load address in a successive load, the secret information stored in `bp` can be encoded into the cache.

```

1 ulong ptrace_get_debugreg(struct task_struct *tsk, int n)
2 {
3     struct thread_struct *thread = &tsk->thread;
4     ulong val = 0;
5
6     if (n < HBP_NUM) {
7         struct perf_event *bp = thread->ptrace_bps[n];
8         if (bp)
9             val = bp->hw.info.address;
10    }
11    ...
12 }
```

Listing 1.1: Linux Kernel: Original Spectre-PHT Vulnerability in Ptrace Function

Dianzhang Chen proposed a fix for this vulnerability, which is shown in Listing 1.2. The sanitized input is highlighted in green, while the insecure input is highlighted in red. The additional speculative load hardening at line 8 solves the problem by masking

the array index during transient execution, thereby preventing an arbitrary speculative out-of-bounds read at line 9. This correct fix landed in the upstream kernel¹.

```

1  ulong ptrace_get_debugreg(struct task_struct *tsk, int n)
2  {
3      struct thread_struct *thread = &tsk->thread;
4      ulong val = 0;
5      int index = n;
6
7      if (n < HBP_NUM) {
8          index = array_index_nospec(index, HBP_NUM);
9          struct perf_event *bp = thread->ptrace_bps[index];
10         if (bp)
11             val = bp->hw.info.address;
12     }
13     ...
14 }
```

Listing 1.2: Linux Kernel: Fixed Ptrace Function

As bug fixes usually get backported from the upstream kernel into the stable kernels, Chen’s changes also landed in the stable tree². Because the original fix shown in Listing 1.2 triggered a compiler warning, the developer who backported the change corrected the warning by slightly adopting the code. The adopted version of the fix is shown in Listing 1.3. Swapping lines 8 and 9 eliminated the compiler warning, but mistakenly reintroduced the Spectre-PHT vulnerability³, as the load hardening is now done after the load. Therefore, the array access at line 8 again uses unsanitized input, indicated by the red background color, allowing a speculative out-of-bounds read. Later the correct fix has been re-applied to the stable tree. By using formal methods such a mistake could quite likely have been detected in the first place.

```

1  ulong ptrace_get_debugreg(struct task_struct *tsk, int n)
2  {
3      struct thread_struct *thread = &tsk->thread;
4      ulong val = 0;
5      int index = n;
6
7      if (n < HBP_NUM) {
8          struct perf_event *bp = thread->ptrace_bps[index];
9          index = array_index_nospec(index, HBP_NUM);
10         if (bp)
11             val = bp->hw.info.address;
12     }
13 }
```

Listing 1.3: Linux Kernel: Reintroduced Spectre-PHT Vulnerability in Ptrace Function

¹Upstream commit: 31a2fbb390fee4231281b939e1979e810f945415

²Stable commit: d1ba61ae4be5e5a5727e303c827591517b6188bb

³CVE: <https://www.cvedetails.com/cve/CVE-2019-15902/>

1.1.2 Microsoft Visual C/C++ Compiler: Missing Spectre Mitigations

```
1 if (x < size) {  
2     v &= a2[a1[x] * 512];  
3 }
```

Listing 1.4: Standard Spectre-PHT Ex.

```
1 if (x < size) {  
2     v &= a2[a1[x << 1] * 512];  
3 }
```

Listing 1.5: Shift Left Spectre-PHT Ex.

The Microsoft Visual C/C++ compiler got support for automatically instrumenting code which might be vulnerable to Spectre-PHT [36]. Microsoft advised all developers, whose software is potentially affected, to recompile their code with the new `/Qspectre` switch enabled. The compiler will then automatically insert speculation barriers where necessary to protect against transient execution attacks. A typical example of a Spectre-PHT vulnerability is depicted in Listing 1.4. As shown by Paul Kocher [27], the compiler correctly detects the vulnerability in this example and inserts a speculation barrier before the load. To reduce the performance impact of the mitigation, the compiler doesn't blindly add speculation barriers everywhere but instead relies on static analysis to find vulnerable patterns in the code.

To test the effectiveness of the Spectre-PHT mitigation, Paul Kocher analyzed 15 variants of the standard example shown in Listing 1.4. Just two out of 15 vulnerable variants were correctly mitigated [27]. Only examples which closely resemble the standard example are correctly identified by the compiler, other examples such as shown in Listing 1.5 are missed. The only difference between the shift left and the standard example is, that an additional left shift is applied to the index variable `x`. Paul Kocher concluded that developers and users cannot rely on the current Spectre-PHT mitigation of MSVC, as the speculation barriers are only effective if applied to all vulnerable code patterns in a process [27]. More powerful analysis tools such as software model checking could help to identify many of the missed Spectre-PHT vulnerabilities, but also help to detect unnecessary mitigations to minimize the performance impact.

1.2 Problem Statement

In the scope of this thesis an automatic, static binary analysis tool for finding transient execution vulnerabilities in binary programs has been implemented. The transient execution attacks of interest are Spectre style attacks [7, 28], more precisely the following two Spectre variants: Spectre-PHT and Spectre-STL.

- First, an intermediate representation (IR) is defined. The syntax and semantics of the IR is designed to capture all the relevant behavior of different RISC and CISC target architectures, such as ARM or x86-64, while still being general enough to ease the development of the tool and its underlying theory. This allows us to transcompile binaries of different target architectures into one generic representation and perform the analysis on top of the generic representation.

- Additionally, the semantics of the IR captures the relevant transient execution behavior of modern CPUs. As Spectre-PHT exploits mis-predicted branching decisions, the semantics of the branching instruction allows mis-prediction and thereby continues the execution on the wrong path for some bounded number of steps before the mis-prediction is finally resolved. Similarly for Spectre-STL, which allows to speculatively by-pass store operations and thereby giving access to stale data during transient execution.
- Furthermore, different microarchitectural components, such as cache or branch target buffer, are formalized. Microarchitectural components allow an adversary to construct a message channel for leaking sensitive information obtained during transient execution. The formalization of the components are independent from each other, such that the components can selectively be included or excluded from the analysis, depending on the concrete threat model. For instance, disabling hyperthreading or preventing processes of distinct trust boundaries to be scheduled on the same hyperthreading siblings⁴ would allow to exclude some microarchitectural components from the analysis, thereby increasing the analysis precision for the specific threat model in use.
- Finally, an analysis tool based on the approach of bounded model checking (BMC) is implemented and evaluated. The result of this thesis is an extensible semantics and a static binary analysis tool that detects Spectre-PHT and Spectre-STL leaks in binary programs.

1.3 Methodological Approach

- First, an extensive literature study must be conducted to gain a good understanding of existing transient execution attacks and how the speculative behavior of modern CPUs, different microarchitectural components and other low-level bits are exactly related to each other. Additionally, the already existing static analysis tool called “Spectector” [21] for detecting transient execution bugs needs to be analyzed and understood, as this thesis is based on the ideas of Spectector. As part of the first step, a comprehensive set of examples for Spectre-style transient execution bugs should be developed.
- The next step is to define the semantics of the IR including the transient execution behavior of modern CPUs as well as different microarchitectural components. The usefulness of the transient execution semantics needs to be validated against the previously generated example collection.
- Furthermore, a hyperproperty for secure speculative information flow needs to be defined. Meaning that if the hyperproperty holds, the binary program under

⁴Coscheduling: scheduling in control groups <https://lwn.net/Articles/764482/>

evaluation is free from Spectre-PHT and/or Spectre-STL leaks. Similar as in Spectector, this hyperproperty will be denoted as “Speculative Non-Interference” (SNI) [21].

- Based on the previously defined transient execution semantics and SNI hyperproperty, a static binary analysis tool should be implemented. The analysis tool should be based on the technique of bounded model checking (BMC), a well-known and strong technique for catching software bugs [26]. The SNI hyperproperty should be implemented by means of self-composition [6], an approach to reduce the secure information flow problem into a safety verification problem.
- After implementing the tool, its detection capabilities should be validated against our own set of examples as well as the well-known Kocher examples [27]. As Spectector was also validated against Kocher’s examples, we can directly compare the detection capabilities of both tools based on these examples. For reproducibility the pre-compiled Spectector benchmarks⁵ should be used. Furthermore, the strengths and weaknesses of our tool should be compared to similar approaches, namely Spectector [21] and “A Formal Approach to Secure Speculation” [10].
- Finally, a small case study should be conducted by running the tool on small- or mid-size real-world programs. As part of this case study different performance metrics, such as run-time or memory consumption as well as the number of detected vulnerabilities in relation to the unwinding bound should be measured. In another experiment the number of detected vulnerabilities in relation to a given time-bound should be evaluated. Inspired by SpecFuzz [38] some potential candidates for the case study are: the OpenSSL/LibreSSL library, the Brotli compression library or the LibYAML parsing library.

1.4 Notation

1.4.1 Set

A set is an unordered collection of elements of same types. We write a set of elements as $\{e_0, e_1, \dots, e_{n-1}\}$ and the empty set as \emptyset . A set $\{e_0, \dots, e_{n-1}\}$ is of type \mathcal{T}^* with $e_i \in \mathcal{T}$ for all $0 \leq i < n$.

1.4.2 Sequence

A sequence is a collection of elements of same types. In contrast to the set, a sequence of elements has a particular order. We write a sequence of elements as $[e_0, e_1, \dots, e_{n-1}]$ and the empty sequence as $[\]$. A sequence $[e_0, \dots, e_{n-1}]$ is of type $[\mathcal{T}]$ with $e_i \in \mathcal{T}$ for all $0 \leq i < n$. The size of a sequence S is given by $|S| \in \mathbb{N}_0$. The i -th element of a sequence S , with $0 \leq i < |S|$, can be accessed by $S(i)$. For instance, let X be a list of

⁵<https://github.com/spectector/spectector-benchmarks/tree/master/target>

numbers $[1, 2, 3]$, then we have $|X| = 3$ and $X(0) = 1$. In the remainder of this thesis, list is used as a synonym for sequence.

1.4.3 Tuple

A tuple is a collection of elements of different types. We write a n -ary tuple of elements as $\langle e_1, e_2, \dots, e_n \rangle$ or (e_1, e_2, \dots, e_n) . The later notation is mainly used for ease of reading in case of nested tuples, like for example $\langle x, (y, z) \rangle$. A tuple $\langle e_1, \dots, e_n \rangle$ is of type $\langle \mathcal{T}_1, \dots, \mathcal{T}_n \rangle$ with $e_i \in \mathcal{T}_i$ for all $1 \leq i \leq n$.

1.4.4 Sequence Builder

We adopt the set-builder notation to sequences and write it as:

$$\left[\underbrace{f(x)}_{\text{transform function}} \mid x \in \underbrace{S}_{\text{input sequence}} \wedge \underbrace{p(x)}_{\text{filter predicate}} \right]$$

Each element x of the input sequence S which satisfies the predicate $p(x)$ will be included in the resulting sequence. Additionally, if a transform function is given, the transformation will be applied to all accepted elements. In contrast to the set-builder notation, the sequence-builder is order preserving. For instance, let X be a list of numbers $[1, -1, 1, -2, 2, 0]$, then $[2x \mid x \in X \wedge x \geq 0]$ will return a list $[2, 2, 4, 0]$.

1.4.5 Sequence Constructor

We let $x :: S$ denote a sequence constructor which based on the sequence $S \in [\mathcal{T}]$ and element $x \in \mathcal{T}$ yields a new sequence with x prepended. Additionally, we let $S \triangleleft x$ denote a sequence constructor which appends the element x to sequence S , hence providing stack-like behavior. We allow to write $x :: y :: S$ in place of $x :: (y :: S)$ respectively $S \triangleleft x \triangleleft y$ in place of $(S \triangleleft x) \triangleleft y$. For instance, let X be a sequence $[2, 3]$ then $1 :: X$ will give the sequence $[1, 2, 3]$.

1.4.6 Sequence Concatenation

We define $S_1 \bowtie S_2$ as the concatenation of two sequences $S_1, S_2 \in [\mathcal{T}]$ such that $[x_1, \dots, x_m] \bowtie [y_1, \dots, y_n]$ yields the sequence $[x_1, \dots, x_m, y_1, \dots, y_n]$. We allow to write $S_1 \bowtie S_2 \bowtie S_3$ in place of $S_1 \bowtie (S_2 \bowtie S_3)$.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Intermediate Representation

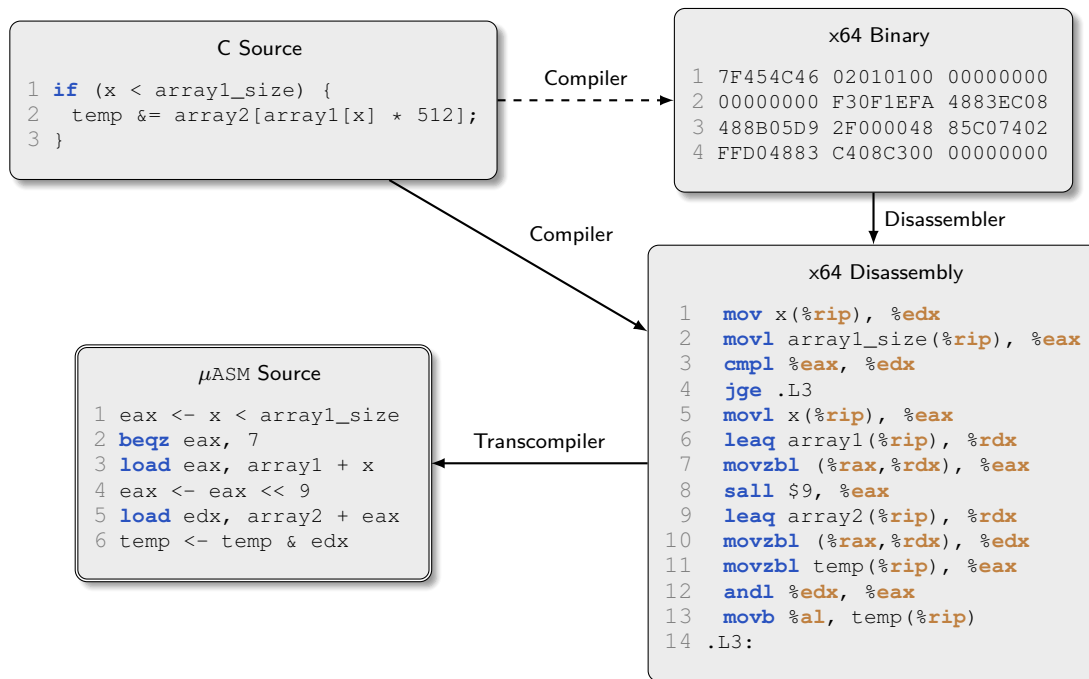


Figure 2.1: From Binary to μ ASM

In this chapter we define the syntax and default semantics of a simplified assembly language called μ ASM. Our definition of μ ASM closely follows the μ ASM definition from the Spectector paper [21].

The language is designed as an intermediate representation (IR) for analysis purposes and only specifies a limited number of basic instructions. Complex instructions, as commonly found in CISC architectures, can be represented by a combination of one

or more basic instructions. The language allows for complex expressions, useful for example when translating x86 address computations like `mov (%esi, %ebx, 4), %edx` to the corresponding μ ASM instruction `load edx, esi+4*ebx`. Additionally the language supports an arbitrary number of registers, hence simplifying the translation from different target architectures with varying number of registers.

Disassemblies available in x86-64, ARM or another assembly language can simply be transcompiled into the corresponding μ ASM program. Figure 2.1 depicts how the translation could look like. The disassembly can be generated from an existing binary using a suitable disassembler or directly be emitted by the compiler. The transcompiler then translates the disassembly into a μ ASM program, which can then be fed into the analyzer.

2.1 Syntax

The μ ASM language is defined as follows [21]:

Registers	$x \in \mathcal{R}$
Values	$n, \ell \in \text{Word}$
Expressions	$e ::= n \mid x \mid \ominus e \mid e \oplus e$
Instructions	$\mathcal{I} ::= x \leftarrow e \mid x \stackrel{\ell}{\leftarrow} e \mid \text{load } x, e \mid \text{store } x, e \mid$ $\text{jmp } \ell \mid \text{beqz } x, \ell \mid \text{skip} \mid \text{spbarr} \mid \text{obs}$
Program	$\rho = [\mathcal{I}_1, \dots, \mathcal{I}_k]$

μ ASM operates on a register-based machine. *Word* corresponds to the machine word of the concrete instruction set and usually has a size of 32 or 64 bits in modern computer architectures. \mathcal{R} is a finite set of registers, where each register is identified by a unique name. A program ρ is a finite sequence of instructions, such that each instruction \mathcal{I}_i is labeled by an address $i \in \text{Word}$. For a program location $pc \notin [1, k]$ we define $\rho(pc) = \perp$.

$x \leftarrow e$ denotes an assignment of expression e to register x . $x \stackrel{\ell}{\leftarrow} e_1$ is a conditional assignment, meaning that expression e_1 is only assigned to register x if e_2 is different from 0, otherwise the assignment is skipped. `load x, e` loads the memory content from address e into register x . `store x, e` stores the value of register x at memory address e . `jmp ℓ` denotes an unconditional branch which jumps to program location ℓ . `beqz x, ℓ` is a conditional branch, meaning that it only jumps to program location ℓ if register x is equal to 0, otherwise the jump is skipped. Both branch instructions are currently limited to static branch targets. `spbarr` is a speculation barrier, meaning that speculative execution will be stopped when hitting this instruction. `obs` is a pseudo instruction denoting that an adversary can leak the program observations when reaching this instruction.

2.2 Default Semantics

2.2.1 Configuration

The derivation relation \longrightarrow works on a configuration $\langle \varphi, \sigma, pc \rangle$, where $pc \in \text{Word}$ is the program counter corresponding to the program location of the current instruction, $\varphi \in \mathcal{R} \rightarrow \text{Word}$ is the current register assignment and $\sigma \in \text{Word} \rightarrow \text{Word}$ is the current memory state.

Our definition of equality relies on the concept of low- and high-equivalence, a well-known concept from information-flow analysis [45]. Configurations are divided into low- and high-security components, where low-security components may be known to the adversary but high-security components are meant to stay secret. Two configurations are considered low-equivalent if their low-security components are equivalent. As the adversary can only observe low-security components, low-equivalent configurations are indistinguishable for the adversary. Let the pair $\mathcal{L} = (\mathcal{R}_L, \mathcal{A}_L)$ be a *security policy* consisting of a finite set of low registers $\mathcal{R}_L \subseteq \mathcal{R}$ and low memory addresses $\mathcal{A}_L \subseteq \text{Word}$. Two configurations are low-equivalent in respect to \mathcal{L} , if and only if the register assignments φ_1, φ_2 agree on the values of the low registers \mathcal{R}_L and the memory states σ_1, σ_2 agree on the values of the low memory addresses \mathcal{A}_L .

$$\langle \varphi_1, \sigma_1, pc_1 \rangle \sim_{\mathcal{L}} \langle \varphi_2, \sigma_2, pc_2 \rangle \equiv \forall r \in \mathcal{R}_L. \varphi_1(r) = \varphi_2(r) \wedge \\ \forall a \in \mathcal{A}_L. \sigma_1(a) = \sigma_2(a)$$

2.2.2 Trace

A trace π is an execution starting at an initial configuration with program counter equal to 1 and terminating in a final configuration with program counter equal to pc' , such that $\rho(pc') = \perp$.

$$\pi \equiv \langle \varphi, \sigma, 1 \rangle \longrightarrow^* \langle \varphi', \sigma', pc' \rangle$$

2.2.3 Expression Evaluation

All expressions are evaluated given a register assignment φ .

$$\llbracket n \rrbracket \varphi = n \quad \llbracket x \rrbracket \varphi = \varphi(x) \quad \llbracket \ominus e \rrbracket \varphi = \ominus \llbracket e \rrbracket \varphi \quad \llbracket e_1 \oplus e_2 \rrbracket \varphi = \llbracket e_1 \rrbracket \varphi \oplus \llbracket e_2 \rrbracket \varphi$$

Where the unary operator \ominus can be any of:

- Arithmetic operators: `neg`
- Bit-wise operators: `not`
- Extension operators: `sxt`, `zxt`

Where the binary operator \oplus can be any of:

- Arithmetic operators: add, sub, mul, udiv, urem, srem, smod
- Bit-wise operators: and, or, xor
- Shift operators: shl, ash, lshr
- Comparison operators: ule, ult, uge, ugt, sle, slt, sge, sgt

2.2.4 Instruction Evaluation

$$\begin{aligned} \text{SKIP} & \frac{\rho(pc) = \text{skip}}{\langle \varphi, \sigma, pc \rangle \longrightarrow \langle \varphi, \sigma, pc + 1 \rangle} \\ \text{BARRIER} & \frac{\rho(pc) = \text{spbarr}}{\langle \varphi, \sigma, pc \rangle \longrightarrow \langle \varphi, \sigma, pc + 1 \rangle} \\ \text{OBSERVE} & \frac{\rho(pc) = \text{obs}}{\langle \varphi, \sigma, pc \rangle \longrightarrow \langle \varphi, \sigma, pc + 1 \rangle} \\ \text{ASSIGN} & \frac{\rho(pc) = x \leftarrow e \quad \varphi' = \varphi[x \mapsto \llbracket e \rrbracket \varphi]}{\langle \varphi, \sigma, pc \rangle \longrightarrow \langle \varphi', \sigma, pc + 1 \rangle} \\ \text{CONDASSIGNAPPLIED} & \frac{\rho(pc) = x \stackrel{e_2}{\leftarrow} e_1 \quad \llbracket e_2 \rrbracket \varphi \neq 0 \quad \varphi' = \varphi[x \mapsto \llbracket e_1 \rrbracket \varphi]}{\langle \varphi, \sigma, pc \rangle \longrightarrow \langle \varphi', \sigma, pc + 1 \rangle} \\ \text{CONDASSIGNSKIPPED} & \frac{\rho(pc) = x \stackrel{e_2}{\leftarrow} e_1 \quad \llbracket e_2 \rrbracket \varphi = 0}{\langle \varphi, \sigma, pc \rangle \longrightarrow \langle \varphi, \sigma, pc + 1 \rangle} \\ \text{LOAD} & \frac{\rho(pc) = \text{load } x, e \quad a = \llbracket e \rrbracket \varphi \quad \varphi' = \varphi[x \mapsto \sigma(a)]}{\langle \varphi, \sigma, pc \rangle \longrightarrow \langle \varphi', \sigma, pc + 1 \rangle} \\ \text{STORE} & \frac{\rho(pc) = \text{store } x, e \quad a = \llbracket e \rrbracket \varphi \quad \sigma' = \sigma[a \mapsto \varphi(x)]}{\langle \varphi, \sigma, pc \rangle \longrightarrow \langle \varphi, \sigma', pc + 1 \rangle} \\ \text{JUMP} & \frac{\rho(pc) = \text{jmp } \ell}{\langle \varphi, \sigma, pc \rangle \longrightarrow \langle \varphi, \sigma, \ell \rangle} \\ \text{BRANCHTAKEN} & \frac{\rho(pc) = \text{beqz } x, \ell \quad \varphi(x) = 0}{\langle \varphi, \sigma, pc \rangle \longrightarrow \langle \varphi, \sigma, \ell \rangle} \\ \text{BRANCHNOTTAKEN} & \frac{\rho(pc) = \text{beqz } x, \ell \quad \varphi(x) \neq 0}{\langle \varphi, \sigma, pc \rangle \longrightarrow \langle \varphi, \sigma, pc + 1 \rangle} \end{aligned}$$

Speculative Execution

Modern processors make use of speculative execution to increase their overall performance. The lower the mis-prediction rate is, the greater the possible performance gain is. Speculative execution is made possible by the so-called out-of-order execution where instructions can be evaluated independent from the program order. Meaning that the processor doesn't have to wait until all operands of an executed instruction are available before it can continue with the next one, but instead the processor can eagerly execute succeeding instructions.

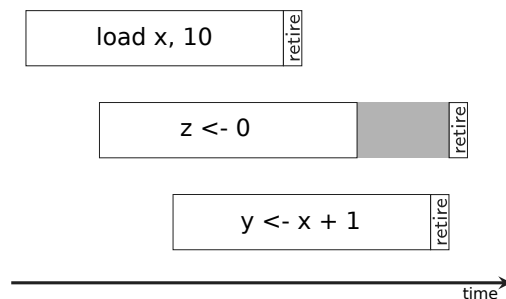


Figure 3.1: Out-of-order Execution and Retirement

During the execution of out-of-order instructions, the state of all instructions is collected in the reorder buffer. The changes are committed to registers and memory once the instructions are retired. While instructions may be executed out-of-order, the retirement happens in program order [47]. An example of this is shown in Figure 3.1. Suppose that we have a program `load x, 10; y ← x + 1; z ← 0`. Then `z ← 0` can be executed before the assignment to `y` instead of waiting until the memory load is done. The retirement of `z ← 0` happens after the assignment to `y` as indicated by the program order. The reorder buffer also keeps track of additional information, such as if an instruction has been speculatively executed and what the prediction was. When

retiring an instruction which caused the speculation, all corresponding buffer entries depending on the prediction are either committed or discarded [47].

Upon the retirement of an instruction the changes become visible on the architectural level, for example by applying the computed results to the memory. Changes which are discarded become not visible on the architectural level. In addition to the architectural state, which consists mainly of registers and memory, there is the microarchitectural state which consists of caches, buffers and other hidden processor internal state. While discarded modifications don't alter the architectural state, they may still have impacts on the microarchitectural state, such as changing the state of the cache during a memory fetch. We have that one architectural state maps to many microarchitectural states, which is exactly the cause of speculative side-channel attacks [33].

3.1 Transient Execution Attacks

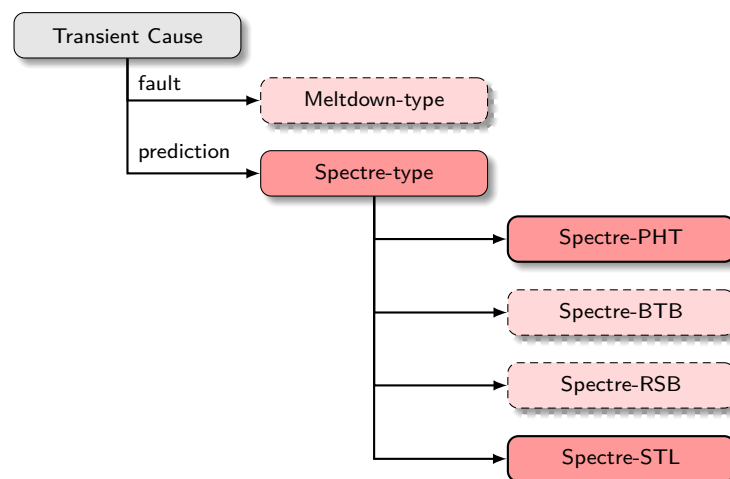


Figure 3.2: Simplified Transient Execution Attack Classification Tree [7]

Canella et al. [7] did a systematic evaluation of all currently known transient execution attacks and defenses. Their paper defines a consistent naming scheme as well as a comprehensive classification tree for transient execution attacks. In Figure 3.2 a simplified version of their classification tree is shown. All transient execution attacks have in common, that they exploit transient execution to leak sensitive information through microarchitectural covert channels. Based on the cause of the transient execution, the attacks can be divided into Meltdown-type attacks and Spectre-type attacks [7].

Meltdown-type attacks exploit the fact that exceptions are only raised upon the retirement of faulting instructions [7]. This means that there exists a small time span between causing and raising an exception, in which succeeding instructions in the execution pipeline can use temporary computation results of the soon to be retired faulting instruction, thereby allowing the transiently executed instructions to operate on values

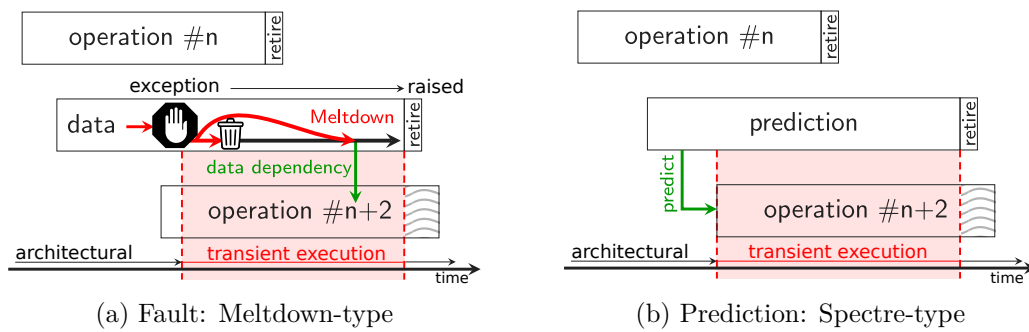


Figure 3.3: Cause of Transient Execution (adopted from [18])

resulting from e.g. unauthorized data accesses, as depicted in Figure 3.3a. Once the faulting instruction gets finally retired, the transiently computed results are discarded and the exception is raised. Microarchitectural covert channels allow to leak the sensitive information obtained during transient execution into the architectural state [28]. For example, the original Meltdown attack [30] circumvents the user/supervisor authorization check of virtual memory pages. Virtual memory pages with supervisor bit set belong to the OS kernel and should therefore only be accessible from within kernel/supervisor mode, accessing them from within user mode should result in a page fault instead. As it turned out [30], the delay between the unauthorized read and the resulting page fault allows an attacker to perform unauthorized reads of arbitrary kernel-memory locations from within user mode. Meltdown mitigations required changes across OS kernels, hypervisors as well as CPU microcode implementations. For a detailed explanation of all the different Meltdown attacks and mitigations we refer to [7].

Spectre-type attacks exploit the speculative behavior of modern CPUs [7, 28]. Instead of waiting for the exact result, modern CPUs try to predict the most likely outcome and transiently execute succeeding instructions in advance using the predicted result, as depicted in Figure 3.3b. The prediction can be control- as well as data-flow related, for example the possible outcome and jump target of a branch instruction or data-dependencies between memory instructions. Once the exact result finally becomes available, the prediction will be verified. If the prediction was correct, the transiently computed results are committed and will therefore become visible on the architectural level. If the prediction was wrong, the transiently computed results are discarded, meaning that they never become visible on the architectural level. But discarding the transiently computed results doesn't completely revert all changes on the microarchitectural level and therefore some effects of the transient execution will remain [7], for example in the cache. While this doesn't influence the correctness of computation on the architectural level, it opens an interesting attack vector for side-channel attacks targeting the microarchitectural level. By mis-training diverse prediction units of the CPU, an attacker can more or less choose the instructions which are transiently executed. Thereby, the transiently executed instructions can reveal sensitive information that is normally not accessible to the attacker. As with Meltdown, microarchitectural covert

channels allow to leak the sensitive information available during transient execution into the architectural state [28].

As shown in Figure 3.2 there are currently four known classes of Spectre-style attacks. Only Spectre-PHT and Spectre-STL are of interest for this thesis and will therefore be explained in more detail in the following section. Spectre-BTB [28] and Spectre-RSB [29, 31] exploit the CPU's branch target buffer (BTB) respectively return stack buffer (RSB) to transiently redirect the control flow to arbitrary program locations. But given that our mechanism doesn't yet cover indirect jumps, we cannot deal with them. Therefore, we ignore Spectre-BTB and Spectre-RSB for the time being.

3.1.1 Attack Phases

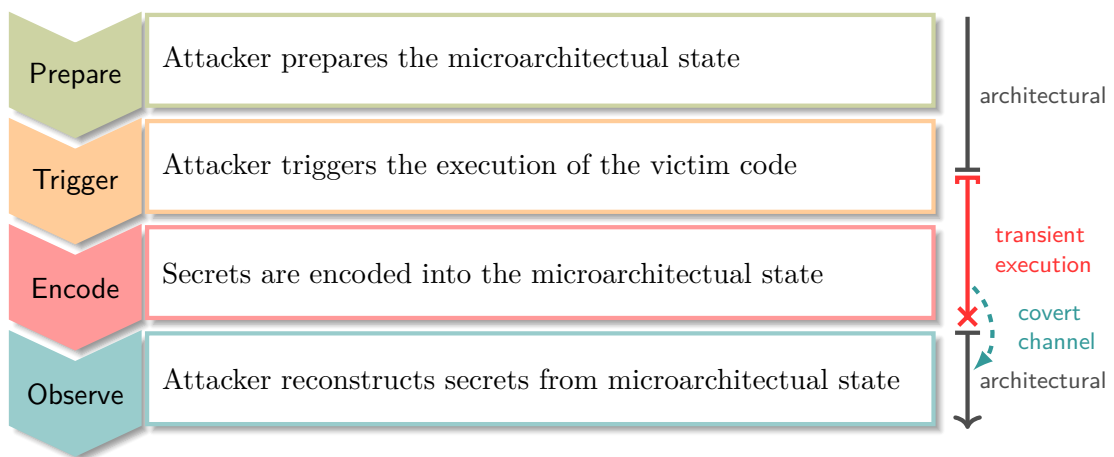


Figure 3.4: Four Phases of a Transient Execution Attack

As shown in Figure 3.4, a transient execution attack can be divided into four phases [7]: (i) The adversary prepares the microarchitectural state, for example by flushing the cache and training the branch predictor. Thereby the microarchitectural components are put into a well-defined state required for the later reconstruction of encoded secret information during the final observe phase. (ii) The adversary triggers the execution of the victim code. At some point the transient execution will start. (iii) During transient execution, secret information is encoded into the microarchitectural state, for example by interpreting the secrets as memory locations and loading them into the cache. At some point the CPU will detect the mis-speculation and consequently rollback the transient execution. As the rollback doesn't affect microarchitectural components such as the cache, all the previously encoded secrets will survive the rollback. (iv) The adversary reconstructs the previously encoded secret information from the microarchitectural state, for example by means of a cache timing-attack as explained in Section 4.1.

3.1.2 Spectre-PHT

Spectre-PHT [7, 28], also known as Spectre variant 1 or Bounds Check Bypass (BCB), is a Spectre-style attack which exploits mis-predicted branching decisions of conditional branches. The branch predictor uses the Pattern History Table (PHT) to decide if a conditional branch should be taken or not, based on past branching decisions [47]. By training the PHT an adversary can influence the control-flow during transient execution, thereby transiently executing instructions which would normally not be reachable. This allows an attacker to perform arbitrary out-of-bounds reads as well as writes [28], thereby gaining access to sensitive information. Microarchitectural covert channels allow to leak the obtained secret information into the architectural state.

Once the real branch condition finally becomes available, the prediction is verified. In case of a mis-prediction, the transiently computed results are discarded and the instructions of the correct branch are re-executed. Code sequences vulnerable to Spectre-PHT roughly have the following form: (i) branch-based access guard, such as bounds check (ii) read of sensitive information into x (iii) gadget for encoding x into the microarchitectural state.

```

1 void adversary() {
2     // 1. Prepare
3     victim(0); // Train
4     flush(&public_len);
5     spbarr();
6
7     // 2. Trigger
8     victim(7);
9
10    // 4. Observe
11    char secret =
12        cache_decode();
13 }
```

Listing 3.1: Spectre-PHT Adversary

```

1 char* public = "PUBLIC";
2 char* private = "SECRET";
3 uint public_len = 6;
4
5
6 void victim(uint x) {
7     if (x < public_len) {
8         char c = public[x];
9
10        // 3. Encode
11        cache_encode(c);
12    }
13 }
```

Listing 3.2: Spectre-PHT Victim

Listings 3.1 and 3.2 show a small Spectre-PHT example. The victim code implements a typical bounds check to guard access to the `public` array, only indices between 0 and `public_len` are allowed to access it. As `public_len` was previously flushed from cache by the adversary, it needs to be loaded from memory before the branch condition can be evaluated. Thus, the CPU will start to speculate at line 7. Because the adversary has pre-trained the PHT using `victim(0)`, the branch predictor will now predict that the then-branch should be taken. Given that the victim function was triggered with $x = 7$, the transiently executed load at line 8 will read out-of-bounds, thereby loading sensitive data from the subsequent `private` string into variable `c`. Finally, the sensitive value stored in `c` is encoded into the cache at line 11. Once the load of `public_len` is complete, the CPU will detect the mis-speculated branch and resolve it. But as the secret value has already been encoded into the cache during transient execution, the

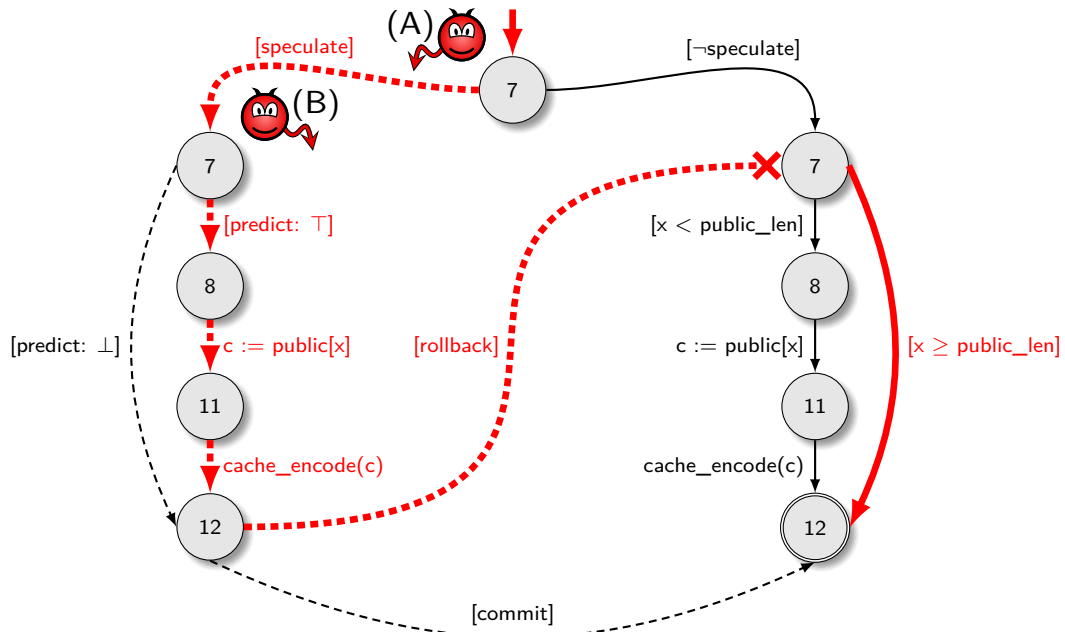


Figure 3.5: Control-Flow Graph and Transient Attack Path of Victim Function

adversary can reconstruct it from the cache.

Figure 3.5 depicts the (transient) execution path of the previously described Spectre-PHT attack. By flushing `public_len` from cache, the adversary can trigger speculative execution at (A). By training the PHT, the adversary can control the branching direction at (B). Given that `c := public[x]` is now reachable without previous bounds-check of index `x`, the adversary can read arbitrary memory content from `public + x` into `c`.

Spectre-PHT can be mitigated by inserting speculation barriers into vulnerable code sequences [25]. Other proposed countermeasures are Index Masking [41] and Speculative Load Hardening (SLH) [9]. Some compilers can automatically insert Spectre-PHT mitigations [8, 36], but as outlined in Section 1.1.2 some Spectre-PHT vulnerabilities may be missed [27].

```

1 void victim(uint x) {
2     if (x < public_len) {
3         char c = public[x];
4         spbarr;
5         cache_encode(c);
6     }
7 }

```

Listing 3.3: Spectre-PHT Mitigation - Speculation Barrier

Speculation barriers prevent the CPU from speculatively executing further instructions, meaning that the transient execution will stop as soon as a barrier instruction

is reached. Listing 3.3 shows the victim code mitigated by speculation barriers. The additional barrier instruction at line 4 prevents the secret information obtained during transient execution from being encoded into the cache.

```

1 void victim(uint x) {
2     if (x < public_len) {
3         uint mask = 0b1111;
4         x &= mask;
5         char c = public[x];
6         cache_encode(c);
7     }
8 }

```

Listing 3.4: Spectre-PHT Mitigation - Index Masking

Index masking applies an additional bit mask to the index variable to protect from arbitrary out-of-bounds reads, thereby reducing the possible index range. Listing 3.4 shows the victim code mitigated by index masking. While this approach drastically reduces the set of available indices during transient execution, it doesn't fully mitigate the problem [41] as the mask is limited to values of the form $2^n - 1 \geq len$. This leaves behind a small set of indices, namely $\{len, \dots, 2^n - 1\}$, which are still vulnerable. For example, in the mitigated victim function shown in Listing 3.2, an adversary could still read out-of-bounds using the indices 6 and 7.

```

1 void victim(uint x) {
2     if (x < public_len) {
3         char c = public[x];
4         char mask = (x >= public_len) ? 0x00 : 0xFF;
5         c &= mask;
6         cache_encode(c);
7     }
8 }

```

Listing 3.5: Spectre-PHT Mitigation - Speculative Load Hardening

SLH uses branchless code to ensure secure speculation [9]. In contrast to speculation barriers, SLH doesn't stop speculative execution but instead prevents the adversary from leaking sensitive information during transient execution. Listing 3.5 shows the victim code mitigated by SLH. The branchless conditional move at line 4 ensures that if the branch condition is mis-predicted, the mask becomes $0x00$ and thus the bitwise-and at line 5 will zero out the sensitive information stored in c . Therefore, c will always be zero when encoded into the cache during transient execution. Note that unlike for conditional branches, the CPU doesn't speculate on the condition of conditional moves, meaning that in our example the mask will only be assigned if $x \geq public_len$ can be evaluated. The advantage of SLH over speculation barriers is, that the Spectre-PHT mitigation using SLH is about 1.77 times faster than using barrier instructions [9]. The reason for the increased performance is, that instructions which don't have a data dependency on the mask can still be speculatively executed in case of SLH, whereas speculation barriers

prevent speculative execution for all instructions. The SLH mitigation requires of course that the masking is compiled into conditional move instructions at machine code level, such as `CMOVCC` in x86. If the compiler accidentally transforms the masking into code with branches then the leakage of sensitive information isn't prevented anymore.

3.1.3 Spectre-STL

Speculation is not limited to control-flow only, instead modern CPUs additionally speculate on data-flow dependencies [47]. Spectre-STL [7, 23, 35], also known as Spectre variant 4 or Speculative Store Bypass (SSB), is a Spectre-style attack which exploits mis-predicted data-flow dependencies during transient execution.

The CPU's memory disambiguator calculates data-flow dependencies between store and load instructions. For Spectre-STL we focus on Read-after-Write dependencies, also called Store-to-Load (STL) dependencies [7]. An STL dependency requires that for a load of memory location x , all preceding stores referencing the same memory location x have been completed before the load is finally executed. Hence, if a load instruction doesn't have any STL dependency on preceding store operations, the load instruction can immediately be executed instead of waiting for the completion of prior store operations. This technique is known as *store bypassing* [47]. For existing STL dependencies another technique called *load forwarding* [47] allows to directly forward values from store to dependent load operations, thereby eliminating memory loads. Both techniques together enable efficient out-of-order execution of memory instructions and may yield significant performance improvements [47].

However, some memory locations may not be known prior the time of execution. For example, an indirect memory store first requires to resolve the target address before the actual store operation can be performed. In such a case, the CPUs' memory disambiguator can only speculate on possible data-flow dependencies [47]. Assuming that there are no STL dependencies between a store and its succeeding loads allows the CPU to speculatively execute succeeding loads in advance, thus bypassing the store. Thereby, the speculatively executed instructions operate on a memory state where the bypassed store operation hasn't been materialized. Meaning that speculatively executed loads, which reference the same memory location as the bypassed store, operate on stale values. Once the target address of the store operation finally becomes known, the predictions are verified. In case of a mis-prediction, the transiently computed results are discarded and the instructions are re-executed.

As demonstrated by researchers from Google Project Zero [23] and Microsoft Security Response Center [35], mis-predictions by the memory disambiguator allow to speculatively bypass store instructions. Stale values may contain sensitive information which otherwise wouldn't be accessible when executed in sequential order. Similar to other Spectre attacks, microarchitectural covert channels allow to leak stale values and thereby disclosing sensitive information during transient execution. Furthermore, operating on stale pointers may cause type confusion¹ during transient execution [23]. Code

¹CWE-843: Access of resource using incompatible type

sequences vulnerable to Spectre-STL roughly have the following form: (i) slow write to memory location x (ii) fast read from memory location x into y (iii) gadget for encoding y into the microarchitectural state.

Spectre-STL can be mitigated by inserting speculation barriers into vulnerable code sequences [25]. Moreover, some CPUs allow to completely disable speculative store bypass, thereby mitigating the problem directly on the microcode-level [25].

```

1
2 void adversary() {
3     // 1. Prepare
4     flush(&data_ptr);
5     spbarr();
6
7     // 2. Trigger
8     victim();
9
10    // 4. Observe
11    char secret =
12        cache_decode();
13 }

```

Listing 3.6: Spectre-STL Adversary

```

1 char data = '#';
2 char* data_ptr = &data;
3
4 void victim() {
5     // Set secret
6     data = choice('A'..'Z');
7
8     // Wipe secret (slow)
9     (*data_ptr) = '#';
10
11    // 3. Encode
12    cache_encode(data);
13 }

```

Listing 3.7: Spectre-STL Victim

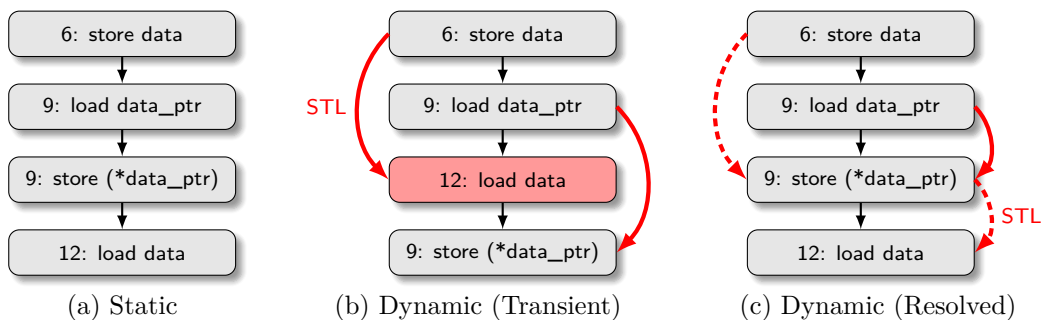


Figure 3.6: Simplified Memory Instruction Sequence of Victim Function

Listings 3.6 and 3.7 show a small Spectre-STL example. The victim chooses a secret value between 'A' and 'Z' and assigns it to `data`. Afterwards, to keep the secret value secure, it's wiped again by setting the global variable `data` back to value '#'. In this case the write is done through pointer indirection, meaning that the target address needs to be loaded before the store can be performed, thereby slowing down the actual secret overwrite. Nonetheless, when taking a look at the static instruction sequence as depicted in Figure 3.6a, `data` should always be equal to '#' when reaching line 12.

As the value of `data_ptr` needs to be loaded from memory (was flushed from cache by the adversary) the CPU will start to speculate at line 9. One possible dynamic instruction sequence including the already known data-dependencies is shown in

Figure 3.6b. Because the memory disambiguator is initially unable to detect the data-dependency between line 9 and 12, the load of line 12 is transiently re-ordered before the store of line 9, meaning that the load speculatively bypasses the store. Thereby, the stale value of `data`, which is the chosen secret value, is loaded and encoded into the cache during transient execution. Once the load of `data_ptr` is complete, the CPU will detect the mis-speculated data-dependency between line 9 and 12 and resolve it like shown in Figure 3.6c. But as the secret value has already been encoded into the cache during transient execution, the adversary can reconstruct it from the cache. Inserting a speculation barrier between line 9 and 12 fixes the vulnerability.

3.2 Setting

We assume that the adversary is in full control of the predictor and microarchitectural state prior to execution. Therefore, we allow the adversary to control for each individual program location if and how to speculate. Furthermore, we assume that the adversary can observe the full microarchitectural state on `obs`. Additionally, we assume that if two microarchitectural states differ the adversary is always able to reconstruct secret information, no matter if this is actually achievable in a real environment.

3.3 Transient Execution Semantics

In this section we extend the non-speculative semantics introduced in Section 2.2 with speculative behavior. As we are interested in finding transient execution bugs, the following semantics only defines transient execution, meaning that speculative execution is limited to mis-predict and rollback. Because speculative execution with correct prediction is indistinguishable² from non-speculative execution, no explicit modeling is required as its behavior is already covered by the non-speculative semantics. For the time being, we limit mis-prediction to branch and load instructions only. Additionally, we assume that all transient executions have the same maximum length $\Omega \in \mathbb{N}_0$. This is motivated by the bounded number of entries in the re-order buffer [47], which gives a natural limit for the length of the transient execution [21]. The re-order buffer can typically hold only a few hundred instructions, for example 224 instructions in case of Intel Skylake CPUs or 192 instructions in case of AMD Ryzen CPUs [32].

3.3.1 Predictor

We introduce an predictor, denoted as \mathcal{Y} , of abstract type \mathcal{P} which models the prepared microarchitectural state given by the adversary. The predicate $\text{speculate}(\mathcal{Y}, pc) \in \mathcal{P} \times \text{Word}$ denotes if speculative execution should be started at program location pc . In real attacks the adversary would for example flush specific values from the cache to trigger the speculative execution. The predicate $\text{taken}(\mathcal{Y}, pc) \in \mathcal{P} \times \text{Word}$ denotes if a branch,

²Indistinguishable in the sense of computation results and microarchitectural effects.

when executed speculatively, should be taken or not. This models the behavior of the branch predictor, more specifically the behavior of the PHT. In real attacks the adversary would for example train the branch predictor accordingly to either take or not take the branch. As we limit speculative execution to branch and load instructions only, we require that $\text{speculate}(\mathcal{Y}, pc)$ is `false` for all other instructions, i.e. $\neg \text{speculate}(\mathcal{Y}, pc)$ if $\rho(pc) \notin \{\text{beqz } x, \ell; \text{store } x, e\}$. The function $\text{speculation-window}(\mathcal{Y}, pc) \in \mathcal{P} \times \text{Word} \mapsto \mathbb{N}_0$ returns the length of the speculation window, which can be any number between 0 and Ω .

3.3.2 Configuration

The speculative derivation relation $\xrightarrow{\text{spec}}$ works on a configuration $\langle \varphi, \sigma, pc, \mathcal{Y}, \Delta \rangle$, where the additional \mathcal{Y} denotes the current state of the predictor and Δ defines the current speculative state. $\Delta = \perp$ denotes an empty speculative state, meaning that instructions are executed non-speculatively. $\Delta = (\omega, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc})$ denotes the current state of an ongoing speculative execution, where $\omega \in \mathbb{N}_0$ is the remaining size of the speculation window, $\widehat{\varphi}$ as well as $\widehat{\sigma}$ correspond to the register assignment and memory state before the speculative execution has been started, and \widehat{pc} denotes the instruction which started the speculative execution.

Let the pair $\mathcal{L} = (\mathcal{R}_L, \mathcal{A}_L)$ be a *security policy* consisting of a finite set of low registers $\mathcal{R}_L \subseteq \mathcal{R}$ and low memory addresses $\mathcal{A}_L \subseteq \text{Word}$. Two configurations are low-equivalent in respect to \mathcal{L} , if and only if the register assignments φ_1, φ_2 agree on the values of the low registers \mathcal{R}_L , the memory states σ_1, σ_2 agree on the values of the low memory addresses \mathcal{A}_L and both predictor states are equal. The reason for including the predictor state is, that the adversary can train the predictor and therefore its state is of low security as well.

$$\begin{aligned} \langle \varphi_1, \sigma_1, pc_1, \mathcal{Y}_1, \Delta_1 \rangle \sim_{\mathcal{L}} \langle \varphi_2, \sigma_2, pc_2, \mathcal{Y}_2, \Delta_2 \rangle &\equiv \forall r \in \mathcal{R}_L. \varphi_1(r) = \varphi_2(r) \wedge \\ &\quad \forall a \in \mathcal{A}_L. \sigma_1(a) = \sigma_2(a) \wedge \\ &\quad \mathcal{Y}_1 = \mathcal{Y}_2 \end{aligned}$$

3.3.3 Trace

A speculative execution trace $\tilde{\pi}$ is an execution starting at an initial configuration with program counter equal to 1 and an empty speculative state, and terminating in a final configuration with program counter equal to pc' , such that $\rho(pc') = \perp$, and an empty speculative state.

$$\tilde{\pi} \equiv \langle \varphi, \sigma, 1, \mathcal{Y}, \perp \rangle \xrightarrow{\text{spec}^*} \langle \varphi', \sigma', pc', \mathcal{Y}, \perp \rangle$$

A non-speculative execution trace $\text{ns}(\tilde{\pi})$ can be obtained by removing all speculative configurations from a given speculative execution trace $\tilde{\pi}$. See Appendix A.1 for a proof that the resulting non-speculation trace is semantically valid.

$$\text{ns}(\tilde{\pi}) \equiv [\langle \varphi, \sigma, pc, \mathcal{Y}, \Delta \rangle \mid \langle \varphi, \sigma, pc, \mathcal{Y}, \Delta \rangle \in \tilde{\pi} \wedge \Delta = \perp]$$

3.3.4 Instruction Evaluation

For each instruction we define its behavior during non-speculative execution (-NS suffix) as well as its behavior during speculative execution (-S suffix).

Both lifting rules perform the default instruction evaluation on top of the speculative one for all instructions not affected by speculation.

$$\text{lift-inst}(i) = \begin{cases} \text{false} & \text{if } i \in \{\text{beqz } x, \ell; \text{store } x, e; \text{spbarr}; \text{obs}\} \\ \text{true} & \text{otherwise} \end{cases}$$

$$\text{SPEC_LIFT-NS} \frac{\text{lift-inst}(\rho(pc)) \quad \langle \varphi, \sigma, pc \rangle \longrightarrow \langle \varphi', \sigma', pc' \rangle}{\langle \varphi, \sigma, pc, \Upsilon, \perp \rangle \xrightarrow{\text{spec}} \langle \varphi', \sigma', pc', \Upsilon, \perp \rangle}$$

$$\text{SPEC_LIFT-S} \frac{\text{lift-inst}(\rho(pc)) \quad \langle \varphi, \sigma, pc \rangle \longrightarrow \langle \varphi', \sigma', pc' \rangle \quad \omega > 0}{\langle \varphi, \sigma, pc, \Upsilon, (\omega, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \rangle \xrightarrow{\text{spec}} \langle \varphi', \sigma', pc', \Upsilon, (\omega - 1, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \rangle}$$

The `obs` instruction is a pseudo instruction and thus invisible to speculation. Therefore, `obs` doesn't affect the speculative state even when executed speculatively.

$$\text{SPEC_OBSERVE} \frac{\rho(pc) = \text{obs}}{\langle \varphi, \sigma, pc, \Upsilon, \Delta \rangle \xrightarrow{\text{spec}} \langle \varphi, \sigma, pc + 1, \Upsilon, \Delta \rangle}$$

The `spbarr` instruction stops the speculative execution by setting the remaining size of the speculation window to 0, thus causing an immediate rollback. When executed non-speculatively, the `spbarr` instruction behaves the same as a `skip` instruction.

$$\text{SPEC_BARRIER-NS} \frac{\rho(pc) = \text{spbarr}}{\langle \varphi, \sigma, pc, \Upsilon, \perp \rangle \xrightarrow{\text{spec}} \langle \varphi, \sigma, pc + 1, \Upsilon, \perp \rangle}$$

$$\text{SPEC_BARRIER-S} \frac{\rho(pc) = \text{spbarr} \quad \omega > 0}{\langle \varphi, \sigma, pc, \Upsilon, (\omega, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \rangle \xrightarrow{\text{spec}} \langle \varphi, \sigma, pc, \Upsilon, (0, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \rangle}$$

The `beqz` instruction may predict the branch decision, meaning that the path of execution is given by the predictor instead of the condition value. In this case the branching decision solely depends on the outcome of the branch predictor, represented by the `taken(Υ, pc)` predicate. If `speculate(Υ, pc)` is `true` during non-speculative execution, a new speculation window of size between 0 and Ω is started. From there on the execution is continued on the speculated path. The current program location, register assignment and memory state are remembered in the speculative state for later rollback. If `speculate(Υ, pc)` is `true` during speculative execution, no new speculation window is started as we don't keep track of nested speculative states. See Appendix A.3 for a proof that this simplification is legit for our purpose. Note that this simplification requires that the instructions' effects are modeled as an unbounded set, such as a cache with

unbounded size. If the set is bounded this simplification is invalid.

$$\text{SPECBRANCH-NS} \frac{\rho(pc) = \text{beqz } x, \ell \quad \neg \text{speculate}(\mathcal{Y}, pc) \quad \langle \varphi, \sigma, pc \rangle \longrightarrow \langle \varphi', \sigma', pc' \rangle}{\langle \varphi, \sigma, pc, \mathcal{Y}, \perp \rangle \xrightarrow{\text{spec}} \langle \varphi', \sigma', pc', \mathcal{Y}, \perp \rangle}$$

$$\text{SPECBRANCHPRED-NS} \frac{\begin{array}{l} \rho(pc) = \text{beqz } x, \ell \\ \text{speculate}(\mathcal{Y}, pc) \quad pc' = \text{ite}(\text{taken}(\mathcal{Y}, pc), \ell, pc + 1) \\ \omega' = \text{speculation-window}(\mathcal{Y}, pc) \end{array}}{\langle \varphi, \sigma, pc, \mathcal{Y}, \perp \rangle \xrightarrow{\text{spec}} \langle \varphi, \sigma, pc', \mathcal{Y}, (\omega', \varphi, \sigma, pc) \rangle}$$

$$\text{SPECBRANCH-S} \frac{\begin{array}{l} \rho(pc) = \text{beqz } x, \ell \\ \neg \text{speculate}(\mathcal{Y}, pc) \quad \omega > 0 \quad \langle \varphi, \sigma, pc \rangle \longrightarrow \langle \varphi', \sigma', pc' \rangle \end{array}}{\langle \varphi, \sigma, pc, \mathcal{Y}, (\omega, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \rangle \xrightarrow{\text{spec}} \langle \varphi', \sigma', pc', \mathcal{Y}, (\omega - 1, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \rangle}$$

$$\text{SPECBRANCHPRED-S} \frac{\begin{array}{l} \rho(pc) = \text{beqz } x, \ell \\ \text{speculate}(\mathcal{Y}, pc) \quad \omega > 0 \quad pc' = \text{ite}(\text{taken}(\mathcal{Y}, pc), \ell, pc + 1) \end{array}}{\langle \varphi, \sigma, pc, \mathcal{Y}, (\omega, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \rangle \xrightarrow{\text{spec}} \langle \varphi, \sigma, pc', \mathcal{Y}, (\omega - 1, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \rangle}$$

The `store` instruction may be speculatively bypassed if the memory disambiguator mis-predicts the Store to Load (STL) dependencies [7, 23]. As before, if `speculate`(\mathcal{Y}, pc) is `true` during non-speculative execution, a new speculation window of size between 0 and Ω is started. As the memory write of the bypassed `store` instruction isn't completed yet, succeeding memory loads from the same memory address return stale values during transient execution. This is reflected by continuing the transient execution with the old memory state in case of speculation, both in `SPECSTOREBYPASS-NS` and `SPECSTOREBYPASS-S`. Finally, bypassed store operations will be materialized when the speculation is resolved.

$$\text{SPECSTORE-NS} \frac{\rho(pc) = \text{store } x, e \quad \neg \text{speculate}(\mathcal{Y}, pc) \quad \langle \varphi, \sigma, pc \rangle \longrightarrow \langle \varphi', \sigma', pc' \rangle}{\langle \varphi, \sigma, pc, \mathcal{Y}, \perp \rangle \xrightarrow{\text{spec}} \langle \varphi', \sigma', pc', \mathcal{Y}, \perp \rangle}$$

$$\text{SPECSTOREBYPASS-NS} \frac{\begin{array}{l} \rho(pc) = \text{store } x, e \\ \text{speculate}(\mathcal{Y}, pc) \quad \omega' = \text{speculation-window}(\mathcal{Y}, pc) \end{array}}{\langle \varphi, \sigma, pc, \mathcal{Y}, \perp \rangle \xrightarrow{\text{spec}} \langle \varphi, \sigma, pc + 1, \mathcal{Y}, (\omega', \varphi, \sigma, pc) \rangle}$$

$$\text{SPECSTORE-S} \frac{\begin{array}{l} \rho(pc) = \text{store } x, e \\ \neg \text{speculate}(\mathcal{Y}, pc) \quad \omega > 0 \quad \langle \varphi, \sigma, pc \rangle \longrightarrow \langle \varphi', \sigma', pc' \rangle \end{array}}{\langle \varphi, \sigma, pc, \mathcal{Y}, (\omega, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \rangle \xrightarrow{\text{spec}} \langle \varphi', \sigma', pc', \mathcal{Y}, (\omega - 1, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \rangle}$$

$$\text{SPECSTOREBYPASS-S} \frac{\rho(pc) = \text{store } x, e \quad \text{speculate}(\mathcal{Y}, pc) \quad \omega > 0}{\langle \varphi, \sigma, pc, \mathcal{Y}, (\omega, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \rangle \xrightarrow{\text{spec}} \langle \varphi, \sigma, pc + 1, \mathcal{Y}, (\omega - 1, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \rangle}$$

The `terminate` rule ensures that the speculative execution doesn't get stuck when reaching the end of the program. Instead, the rule triggers a speculative rollback by

setting the remaining size of the speculation window to 0.

$$\text{SPEC_TERMINATE} \frac{\rho(pc) = \perp \quad \omega > 0}{\langle \varphi, \sigma, pc, \Upsilon, (\omega, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \rangle \xrightarrow{\text{spec}} \langle \varphi, \sigma, pc, \Upsilon, (0, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \rangle}$$

The rollback rule restores the non-speculative state when the speculative execution ends, meaning that the speculation window reaches 0. This is done by re-evaluating the instruction which started the speculative execution based on the saved register assignment and memory state, but this time using the non-speculative semantics. The execution along the correct path is continued from there on.

$$\text{SPEC_ROLLBACK} \frac{\langle \widehat{\varphi}, \widehat{\sigma}, \widehat{pc} \rangle \longrightarrow \langle \varphi', \sigma', pc' \rangle}{\langle \varphi, \sigma, pc, \Upsilon, (0, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \rangle \xrightarrow{\text{spec}} \langle \varphi', \sigma', pc', \Upsilon, \perp \rangle}$$

For the remainder of this thesis, let $\text{rollback}(\Delta)$ be a predicate which evaluates to `true` if the speculative state Δ reached the end of the speculation window and therefore triggers a rollback, or `false` otherwise.

$$\text{rollback}(\Delta) = \begin{cases} \text{true} & \text{if } \Delta = (\omega, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \wedge \omega = 0 \\ \text{false} & \text{otherwise} \end{cases}$$

3.4 Observations

The transient execution semantics as defined in Section 3.3 only covers the architectural level, consisting of registers and memory. In this section we extend the transient execution semantics with an additional microarchitectural state, for capturing the instructions effects on the microarchitectural level. The following definitions are all based on an abstract microarchitectural component. For each concrete microarchitectural component appropriate $\mathcal{K}\text{-init}()$, $\mathcal{K}\text{-equiv}()$ and $\mathcal{K}\text{-effects}()$ functions as well as the microarchitectural state type \mathcal{K} need to be specified. Different implementations of microarchitectural components are described in Chapter 4.

The microarchitectural derivation relation $\xrightarrow{\mu_{\text{arch}}}$ works on an extended configuration $\langle \varphi, \sigma, pc, \Upsilon, \Delta, \kappa \rangle$, where the additional κ defines the microarchitectural state. Let the microarchitectural state κ be of abstract type \mathcal{K} . We define a microarchitectural step rule which evaluates the instructions using the previously defined transient execution semantics and additionally evolves the microarchitectural state depending on the currently executed instruction. Instructions which start a new speculative execution won't have any immediate effects on the microarchitectural state, instead their effects will become visible once the speculation is finally resolved.

$$\mu_{\text{ARCH}} \frac{\langle \varphi, \sigma, pc, \Upsilon, \Delta \rangle \xrightarrow{\text{spec}} \langle \varphi', \sigma', pc', \Upsilon', \Delta' \rangle \quad \kappa' = \begin{cases} \mathcal{K}\text{-effects}(\kappa, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}, \varphi', \sigma', pc') & \text{if } \text{rollback}(\Delta) \text{ with } \Delta = (\omega, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \\ \mathcal{K}\text{-effects}(\kappa, \varphi, \sigma, pc, \varphi', \sigma', pc') & \text{if } \neg \text{rollback}(\Delta) \wedge \neg \text{speculate}(\Upsilon, pc) \\ \kappa & \text{if } \neg \text{rollback}(\Delta) \wedge \text{speculate}(\Upsilon, pc) \end{cases}}{\langle \varphi, \sigma, pc, \Upsilon, \Delta, \kappa \rangle \xrightarrow{\mu_{\text{arch}}} \langle \varphi', \sigma', pc', \Upsilon', \Delta', \kappa' \rangle}$$

The instructions effects on the microarchitectural component are encoded using the \mathcal{K} -effects($\kappa, \varphi, \sigma, pc, \varphi', \sigma', pc'$) function. Based on the current microarchitectural state κ , the current register assignment φ , the current memory state σ , the current program counter pc , the next register assignment φ' , the next memory state σ' and the next program counter pc' , the effects of the executed instruction are determined.

A microarchitectural execution trace π^μ extends a speculative execution trace $\tilde{\pi}$ with the additional microarchitectural state. The execution starts at an initial configuration with an empty³ microarchitectural state κ and terminates in a final configuration with a final microarchitectural state κ' .

$$\pi^\mu \equiv \langle \varphi, \sigma, 1, \mathcal{Y}, \perp, \kappa \rangle \xrightarrow{\mu\text{arch}^*} \langle \varphi', \sigma', pc', \mathcal{Y}', \perp, \kappa' \rangle \quad \text{with } \kappa = \mathcal{K}\text{-init}()$$

As the microarchitectural state isn't directly accessible from the architectural level, a microarchitectural covert channel [16] needs to be used. A microarchitectural covert channel allows to transfer information from the microarchitectural state into the architectural state, thereby making it visible on the architectural level. We use the `obs` instruction to model the information transfer of arbitrary microarchitectural covert channels. Given a microarchitectural execution trace π^μ , let `observations()` project it to a sequence of observations for all executed `obs` instructions.

$$\text{observations}(\pi^\mu) \in \mathcal{K}^* \equiv [\kappa \mid \langle \varphi, \sigma, pc, \mathcal{Y}, \Delta, \kappa \rangle \in \pi^\mu \wedge \rho(pc) = \text{obs}]$$

Let $\mathcal{K}\text{-equiv}(\kappa_1, \kappa_2)$ be a predicate which evaluate to `true` iff both microarchitectural states $\kappa_1, \kappa_2 \in \mathcal{K}$ are equivalent. Then two lists of observations $\mathcal{O}_1, \mathcal{O}_2 \in \mathcal{K}^*$ are equivalent iff their effects on the microarchitectural component are identical.

$$\mathcal{O}_1 \sim_{\text{Obs}} \mathcal{O}_2 \equiv |\mathcal{O}_1| = |\mathcal{O}_2| \wedge \forall i. \mathcal{K}\text{-equiv}(\mathcal{O}_1(i), \mathcal{O}_2(i))$$

Two microarchitectural execution traces π^{μ_1}, π^{μ_2} are observationally equivalent iff their observable effects on the microarchitectural component are identical.

$$\pi^{\mu_1} \approx_{\text{Obs}} \pi^{\mu_2} \equiv \text{observations}(\pi^{\mu_1}) \sim_{\text{Obs}} \text{observations}(\pi^{\mu_2})$$

To define observational equivalence for speculative execution traces, the microarchitectural effects need to be obtained first. Let the function $\mu\text{arch}\text{-trace}()$ map a speculative execution trace $\tilde{\pi}$ into the corresponding microarchitectural execution trace.

$$\mu\text{arch}\text{-trace}(\tilde{\pi}) \equiv [\langle \varphi_i, \sigma_i, pc_i, \mathcal{Y}_i, \Delta_i, \kappa_i \rangle \mid \langle \varphi_i, \sigma_i, pc_i, \mathcal{Y}_i, \Delta_i \rangle \in \tilde{\pi}]$$

$$\text{where } \langle \varphi_i, \sigma_i, pc_i, \mathcal{Y}_i, \Delta_i, \kappa_i \rangle = \begin{cases} \langle \varphi_0, \sigma_0, pc_0, \mathcal{Y}_0, \Delta_0, \kappa_0 \rangle \text{ with } \kappa_0 = \mathcal{K}\text{-init}() & \text{if } i = 0 \\ \langle \varphi_{i-1}, \sigma_{i-1}, pc_{i-1}, \mathcal{Y}_{i-1}, \Delta_{i-1}, \kappa_{i-1} \rangle & \\ \xrightarrow{\mu\text{ARCH}} \langle \varphi_i, \sigma_i, pc_i, \mathcal{Y}_i, \Delta_i, \kappa_i \rangle & \text{otherwise} \end{cases}$$

Two speculative execution traces $\tilde{\pi}_1, \tilde{\pi}_2$ are observationally equivalent iff their observable effects on the microarchitectural component are identical.

$$\tilde{\pi}_1 \approx_{\text{Obs}} \tilde{\pi}_2 \equiv \mu\text{arch}\text{-trace}(\tilde{\pi}_1) \approx_{\text{Obs}} \mu\text{arch}\text{-trace}(\tilde{\pi}_2)$$

³The definition of empty depends on the actual microarchitectural component.

3.5 Speculative Non-Interference

We define a 2-safety hyperproperty called speculative non-interference (SNI), similar to the hyperproperty defined in [21]. A program respects SNI if its speculative execution doesn't leak more information than its non-speculative execution. In other words, information that is leaked during non-speculative execution can also be leaked during speculative execution but not vice versa. More formally, a program respects SNI if two speculative execution traces with low-equivalent initial configurations and indistinguishable non-speculative observations, result in indistinguishable speculative observations.

Definition 1 (SNI) *A program ρ satisfies SNI given security policy \mathcal{L} if and only if*

$$\forall \tilde{\pi}_1, \tilde{\pi}_2. \tilde{\pi}_1^0 \sim_{\mathcal{L}} \tilde{\pi}_2^0 \wedge ns(\tilde{\pi}_1) \approx_{Obs} ns(\tilde{\pi}_2) \implies \tilde{\pi}_1 \approx_{Obs} \tilde{\pi}_2$$

Let the security policy \mathcal{L} be chosen such that it reflects all non-confidential information visible to an attacker [45]. If a program satisfies SNI, then an attacker cannot distinguish between two secret inputs to the program by observing the program's effects on the microarchitectural components which happen during transient execution. Hence, SNI guarantees secure speculative information flow and the absence of transient execution leaks.

We argue that our definition of SNI is technically practical, as we let the program on a processor without speculation be observable equivalent, while checking if the same program executed on a processor with speculation gives adversary distinguishable security sensitive observations. This in turn means that the power of SNI inevitably depends on the underlying transient execution semantics. Speculative behavior of a processor which isn't modeled by the transient execution semantics can give false negatives, meaning that a program leaks while SNI holds. Our current definition of observational equivalence is relatively powerful as it allows an adversary to basically count the number of side effects, which is caused by the strict equivalence of two traces. Instead of forcing strict equivalence a more liberal stuttering equivalence would maybe be more practical in future. A comparison of our definition of SNI to Spectector's definition of SNI is given in Section 7.1.

Microarchitecture

As demonstrated by various different speculative execution attacks, diverse microarchitectural components can be used to encode leaked data, such as: cache [1, 7, 28], DRAM [40], floating-point unit (FPU) [49], AVX unit [46], return stack buffer (RSB) [29] or branch target buffer (BTB) [15]. By using an appropriate microarchitectural covert channel [16], the secret information collected during speculative execution can be transferred from the microarchitectural state into the architectural state, therefore making the secret information visible on the architectural level.

In this chapter some microarchitectural components are formalized. For each component we define an appropriate state as well as functions for manipulating the state. Additionally, we define predicates to compare component states.

Example

```
1 if (x < array1_size) {  
2     y = array1[x];  
3     leak_least_significant_bit(y);  
4 }
```

Listing 4.1: Spectre-PHT Example: Leak Least-Significant Bit of y

For explaining how the microarchitectural components can be used to encode leaked data, we adopt the code shown in Listing 4.1 to each formalized component. We replace the abstract `leak_least_significant_bit(y)` at line 3 with proper instructions, such that the least-significant bit of y is encoded in the component's state.

Assume that the code fragment depicted in Listing 4.1 is part of a function that has an argument x of low security, meaning that x can be controlled by an adversary. The intended behavior of the conditional branch at line 1 is, that the content of `array1` should only be accessible if x lies within the array bounds. Therefore, preventing that sensitive data can be read at line 2.

Unfortunately, during speculative execution the load instruction at line 2 can be executed with $x \geq \text{array1_size}$ by circumventing the bounds check. An adversary can train the branch predictor to speculatively execute the then-branch, thus making it possible to speculatively load sensitive data from (almost) arbitrary memory addresses at line 2. Later when the processor detects the mis-speculation, the modifications on the architectural level will be reverted, but the leaked data that has been encoded in the microarchitectural state will remain.

4.1 Cache

Modern processors make use of caches to reduce the average memory access latency [47]. Given that the main memory usually works on a much lower speed than the processor, the processor always has to wait until data is fetched from or written to main memory. Because of this, recently accessed memory locations are buffered in a small but fast cache memory. Hence, access requests to currently cached memory locations, called *cache hits*, can be satisfied very quickly. Unfortunately, not the whole main memory content fits into the cache. Access requests to uncached memory locations, called *cache misses*, still need to wait until the data is fetched from main memory. Therefore, *cache misses* are at least one order of magnitude slower than *cache hits* [50]. This measurable timing difference can be exploited by cache attacks [20]. In the past a great variety of such cache attacks have been demonstrated:

- **Evict+Time** [39]: First, the victim code is executed and its execution time is measured. Then before executing the victim code a second time, the adversary evicts some of the victim’s memory locations from the cache. Eviction works by accessing memory locations corresponding to the same cache lines as the victim’s memory locations, thus causing that the latter ones are dropped from the cache. By comparing the execution time of both runs, the adversary can check if the victim has accessed previously evicted memory locations. An increase of the execution time is highly likely caused by a *cache miss* of an evicted memory location.
- **Prime+Probe** [39, 50]: First, the adversary fills the cache respectively the specific cache lines of interest. Then after executing the victim code, the adversary checks which of the previously cached memory locations have been replaced by the victim. Based on timing measurement of consecutive memory accesses, the set of replaced memory locations can be determined. Each *cache miss* corresponds to a memory access performed by the victim.
- **Flush+Reload** [22, 52]: First, the adversary clears the cache respectively the specific cache lines of interest using the flush instruction¹. Then after executing the victim code, the adversary checks which memory locations have been reloaded by the victim. Based on timing measurement of consecutive memory accesses, the set of reloaded memory locations can be determined. Each *cache hit* corresponds to a memory access performed by the victim.

¹`cflush` in x86-64 [24]

- **Evict+Reload** [19]: This attack combines the first phase of *Evict+Time* with the second phase of *Flush+Reload*. The adversary first evicts the victim’s memory locations from the cache and then determines the set of reloaded memory locations by timing the memory accesses. Unlike *Flush+Reload*, this attack works even if no flush instruction is available.
- **Flush+Flush** [20]: This attack is another variation of *Flush+Reload*, using cache line flushes instead of consecutive memory loads in the second phase. As with memory accesses, the execution time of flush operations also depends on whether memory locations are currently cached or not. Compared to the other cache attacks, this attack is faster and stealthier² as it doesn’t require any memory accesses.

All the previously mentioned cache attacks have two things in common: (i) The victim alters the prepared cache state by accessing memory locations. (ii) The adversary extracts the differences between the prepared and altered cache states by means of a side-channel attack. Hence, our model should be independent from the underlying cache attack and instead only focus on the accessed memory locations. For example, a cache line is initially empty in case of an *Flush+Reload* attack, but occupied in case of an *Evict+Reload* attack. In the first case a memory load by the victim only fills the corresponding cache line, whereas in the second case a memory load will replace an already filled cache line. Although started from contrary cache line states, both cases end up in the same final cache line state.

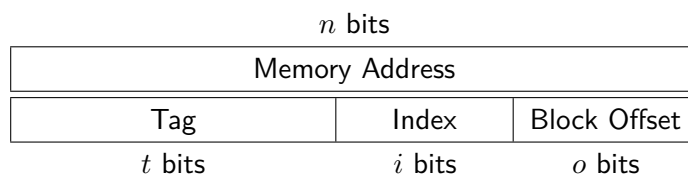


Figure 4.1: From Memory Address to Cache Tag, Index and Block Offset

Caches are organized into a number of fixed size cache lines, each one buffering a non-overlapping portion of the main memory [47]. The size of the cache line defines the granularity of the memory access, meaning that not only the specific bytes which are currently required are put into the cache, but instead the whole memory block of the same size as the cache line is copied from the main memory into the corresponding cache line. Therefore, the cache enables spatial reuse in addition to temporal reuse [47], implying that not only previously accessed memory locations highly likely result in a cache hit but also nearby accessed memory locations. For addressing individual bytes within a cache line the block offset of a memory location, as depicted in Figure 4.1, is used. The number of bits for the block offset depends on the size of the cache line, for example 64 bytes in case of an Intel Skylake CPU³, hence we have $o = \log_2(\text{cache line size})$ where *cache line size* is a power of two in bytes.

²*Flush+Flush* attacks cannot be detected by monitoring cache hits/misses [20].

³<https://www.7-cpu.com/cpu/Skylake.html>

Because the cache can only buffer a small subset of the main memory, sooner or later existing cache lines need to be replaced. The replacement policy dictates which cache line should be evicted from cache when a new one needs to be stored [47]. One of the simplest replacement policies is Random Replacement (RR), which randomly selects a cache line for eviction. Another commonly found replacement policy is Least Recently Used (LRU), which selects the least recently used cache line for eviction and therefore requires additional bookkeeping. Many different replacement policies have been suggested in the literature [13, 43].

Depending on the placement strategy, a memory location can reside in on or multiple different places within the cache. In a direct mapped cache each memory block has exactly one possible cache line, in a set-associative cache each memory block has a fixed set of possible cache lines, denoted as cache set, and in a fully-associative cache each memory block can be stored in any cache line [47]. For direct mapped as well as set-associative caches the index of a memory location, as demonstrated in Figure 4.1, is used to link the memory location to the corresponding cache line respectively cache set. For fully-associative caches the index is omitted. Because multiple memory locations map to the same cache line, the tag as shown in Figure 4.1 is used to check if the already cached memory block corresponds to the requested memory location.

Caching as described above doesn't only apply to data. Instructions also need to be read from the slow main memory, hence the memory access latency is also relevant in this case. Therefore, processors usually have a dedicated instruction cache (I-cache) in addition to the data cache (D-cache) to reduce the average instruction fetch latency [47]. Unfortunately, some cache attacks shown above can also be used for attacking the I-cache in a similar fashion, allowing an adversary to reveal the partial or entire execution flow of the victim [1]. Nonetheless, our cache model only covers the D-cache.

4.1.1 Model

A cache state $\mathcal{C} \subseteq \text{Word}$ is a set of memory locations. In contrast to concrete cache implementations which are typically based on tagged data structures [47], our abstraction keeps track of each individual memory location instead of a multitude of memory locations. Additionally, our model doesn't take cache lines into account, meaning that memory locations are tracked individually instead of blockwise, which may lead to false positives. Cache line awareness can easily be added by applying a bit-mask to all memory locations, such that the block offset is ignored⁴.

Initially the cache is empty.

$$\text{cache-init}() \equiv \emptyset$$

Fetching a memory location into the cache means that the memory location is added to set of cached memory locations.

$$\text{cache-fetch}(\mathcal{C}, a) \equiv \mathcal{C} \cup \{a\} \quad \text{where } a \in \text{Word}$$

⁴Special care needs to be taken for memory accesses across cache line boundaries.

Two cache states $\mathcal{C}_1, \mathcal{C}_2$ are equivalent if both contain the same memory locations.

$$\text{cache-equiv}(\mathcal{C}_1, \mathcal{C}_2) \equiv \mathcal{C}_1 = \mathcal{C}_2$$

4.1.2 Configuration

The derivation relation $\xrightarrow{\text{cache-effects}}$ works on a configuration $\langle \mathcal{C}, \varphi, \sigma, pc, \varphi', \sigma', pc' \rangle$, where \mathcal{C} defines the current state of the cache, φ the current register assignment, σ the current memory state, pc the current program counter, φ' the next register assignment, σ' the next memory state and pc' the next program counter.

4.1.3 Instruction Evaluation

Memory loads as well as memory stores fetch memory content into the cache, thereby making their memory locations visible in the cache. As `store` instructions can be speculatively bypassed, therefore their changes in memory as well as their effects on the cache are ignored until rollback. On rollback, when the write materializes and the mis-predicted STL dependencies are resolved, their effects on the cache will finally become visible. Non-bypassed `store` instructions immediately become visible in the cache. As the μarch semantics already implements this behavior, no special treatment is required for the `store` and `load` instruction and thus both instructions always fetch the memory location a into the cache. All other instructions leave the state of the cache unaffected.

$$\begin{array}{c} \text{CACHeload} \frac{\rho(pc) = \text{load } x, e \quad a = \llbracket e \rrbracket \varphi \quad \mathcal{C}' = \text{cache-fetch}(\mathcal{C}, a)}{\langle \mathcal{C}, \varphi, \sigma, pc, \varphi', \sigma', pc' \rangle \xrightarrow{\text{cache-effects}} \mathcal{C}'} \\ \text{CACHESTORE} \frac{\rho(pc) = \text{store } x, e \quad a = \llbracket e \rrbracket \varphi \quad \mathcal{C}' = \text{cache-fetch}(\mathcal{C}, a)}{\langle \mathcal{C}, \varphi, \sigma, pc, \varphi', \sigma', pc' \rangle \xrightarrow{\text{cache-effects}} \mathcal{C}'} \\ \text{CACHEIGNORE} \frac{\rho(pc) \notin \{\text{load } x, e; \text{store } x, e\}}{\langle \mathcal{C}, \varphi, \sigma, pc, \varphi', \sigma', pc' \rangle \xrightarrow{\text{cache-effects}} \mathcal{C}} \end{array}$$

4.1.4 Example

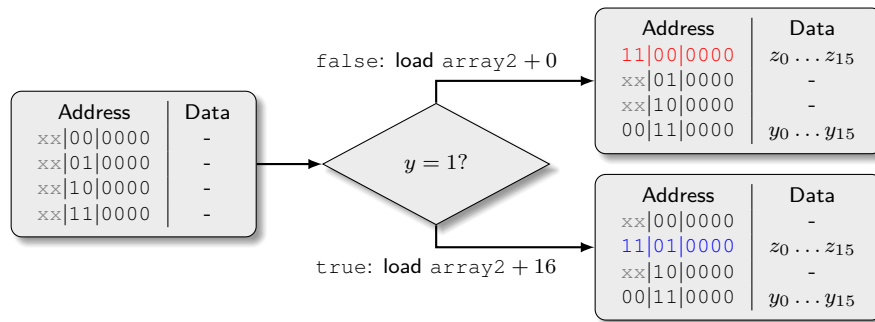
```
1 if (x < array1_size) {
2     y = array1[x];
3     y = y & 1;
4     z = array2[y * 16];
5 }
```

Listing 4.2: Cache Leak (C-Code)

```
1 c <- x < array1_size
2 beqz c, 6
3 load y, array1 + x
4 y <- y & 1
5 load z, array2 + y * 16
```

Listing 4.3: Cache Leak (μASM -Code)

Listings 4.2 and 4.3 adopt the example from the introduction part of Chapter 4. The highlighted lines denote the implementation of `leak_least_significant_bit(y)`. Given a direct mapped cache of 64-byte in size with four cache lines, hence each cache

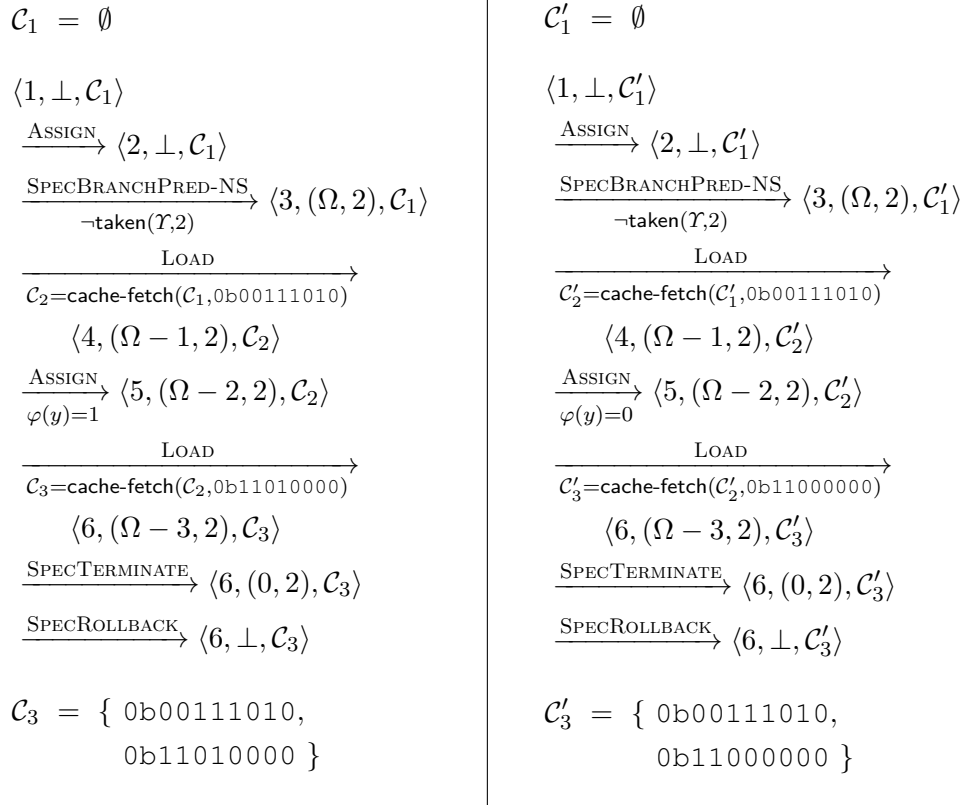
Figure 4.2: Cache State Depending on the Least-Significant Bit of y

line contains 16-byte of data. The four least-significant bits of the memory location correspond to the (byte-)offset within the cache line, the next higher two bits denote the cache line index and the two most-significant bits are the cache tag. For example $0b10110100$ has offset $0b0100$, maps to cache line $0b11$ and has tag $0b10$. The $\times\times$ tag denotes that the cache line is currently unoccupied. We have $x = 10$, $\text{array1_size} = 8$, $\text{array1} = 0b00110000$ and $\text{array2} = 0b11000000$. Assume that the processor speculatively executes the code shown in Listing 4.3 and that the adversary runs a Flush+Reload attack, hence the cache is initially empty.

Because the adversary is in control of x , the load instruction at line 3 can speculatively load sensitive data into register y . This load leaves an irrelevant footprint in the cache, more interesting is the next load. The memory location of the consecutive load at line 5 depends on the value of y . If y is 0 the memory location is $0b11000000$, otherwise if y is 1 the memory location is $0b11010000$. Accessing $0b11000000$ occupies cache line 0 whereas accessing $0b11010000$ occupies cache line 1. This way the least-significant bit of y can be encoded into the cache. The two possible evolutions of the cache are depicted in Figure 4.2.

Finally, the adversary can decode the information from the cache by means of a timing attack. Measuring the execution time of memory loads reveals the least-significant bit of y . If loading $0b11000000$ results in a *cache hit*, indicated by a short delay, but loading $0b11010000$ results in a *cache miss*, indicated by a long delay, then the least-significant bit of y was 0. If exactly the opposite is true, then the least-significant bit of y was 1. Note that for a bit-wise leak as shown in this example, timing a single memory access would already be sufficient [30] to tell if y was 0 or 1, but in general multiple probes are required.

Applying the cache model to the example shown in Listing 4.3 gives two possible cache states \mathcal{C}_3 and \mathcal{C}'_3 which are equivalent non-speculatively but differ speculatively because of the secret-dependent speculative memory load at line 5. The simplified traces of both runs are shown in Figure 4.3.

Figure 4.3: Cache Example Traces for $\text{LSB}(y) = 1$ and $\text{LSB}(y) = 0$

4.2 AVX Unit

The Advanced Vector Extensions (AVX) enhance the x86-64 instruction set architecture with additional vector processing capabilities, including many new instructions and up to 512-bit wide registers [24]. Many applications, such as cryptographic algorithms, can benefit from SIMD⁵ processing. Instead of execution the same operation for each single value one by one, the operation is applied only once to a vector of values.

However, performing vector instructions can consume significant power [17]. Therefore parts of the AVX unit can be turned off by the processor's power management if idling for more than one millisecond [46]. The AVX unit will automatically be powered on again as soon as a vector instruction needs to be executed. But the startup operation takes a while and thus will noticeably delay the execution of the subsequent vector instruction. By measuring the execution time of a vector instruction, an adversary can determine if the AVX unit was previously enabled or disabled. This observable timing difference of vector instructions can be used as a microarchitectural covert channel to transfer one bit of sensitive information at a time [46]. During preparation phase, an

⁵Single Instruction, Multiple Data

adversary can bring the AVX unit into the off state by simply keeping it idle for long enough so that the power management will turn it off.

Motivated by the AVX-based variant of the NetSpectre attack [46], we model the AVX unit for proving the absence of speculative leaks caused by the observable timing differences of vector instructions.

4.2.1 Model

Let $\mathcal{A} \in \{\text{Off}, \text{On}\}$ represent the state of the AVX unit, where **On** signifies that the AVX unit is enabled and **Off** signifies that the AVX unit is disabled.

Initially the AVX unit is disabled.

$$\text{avx-init}() \equiv \text{Off}$$

Enabling the AVX unit sets the state to **On**. Disabling the AVX unit isn't modeled, as its behavior solely depends on the processor's power management as well as the execution/idling time which we don't model.

$$\text{avx-enable}(\mathcal{A}) \equiv \text{On}$$

Two AVX unit states $\mathcal{A}_1, \mathcal{A}_2$ are equivalent if both are either enabled or disabled.

$$\text{avx-equiv}(\mathcal{A}_1, \mathcal{A}_2) \equiv \mathcal{A}_1 = \mathcal{A}_2$$

4.2.2 Configuration

The derivation relation $\xrightarrow{\text{avx-effects}}$ works on a configuration $\langle \mathcal{A}, \varphi, \sigma, pc, \varphi', \sigma', pc' \rangle$, where \mathcal{A} defines the current state of the AVX unit, φ the current register assignment, σ the current memory state, pc the current program counter, φ' the next register assignment, σ' the next memory state and pc' the next program counter.

4.2.3 Instruction Evaluation

Let $\text{requires-avx}(e)$ be a predicate which evaluates to **true** if expression e contains a vector expression or **false** otherwise. The AVX unit is only enabled when executing assignment instructions which contain vector expressions. All other instructions as well as assignment instructions without vector expressions leave the state of the AVX unit unaffected.

$$\text{AVXASSIGNVEC} \frac{\rho(pc) = x \leftarrow e \quad \text{requires-avx}(e) \quad \mathcal{A}' = \text{avx-enable}(\mathcal{A})}{\langle \mathcal{A}, \varphi, \sigma, pc, \varphi', \sigma', pc' \rangle \xrightarrow{\text{avx-effects}} \mathcal{A}'}$$

$$\text{AVXASSIGNNONVEC} \frac{\rho(pc) = x \leftarrow e \quad \neg \text{requires-avx}(e)}{\langle \mathcal{A}, \varphi, \sigma, pc, \varphi', \sigma', pc' \rangle \xrightarrow{\text{avx-effects}} \mathcal{A}}$$

$$\text{AVXIGNORE} \frac{\rho(pc) \neq x \leftarrow e}{\langle \mathcal{A}, \varphi, \sigma, pc, \varphi', \sigma', pc' \rangle \xrightarrow{\text{avx-effects}} \mathcal{A}}$$

4.2.4 Example

```

1 if (x < array1_size) {
2   y = array1[x];
3   if (y & 1) {
4     z = _avx_add(a, b);
5   }
6 }

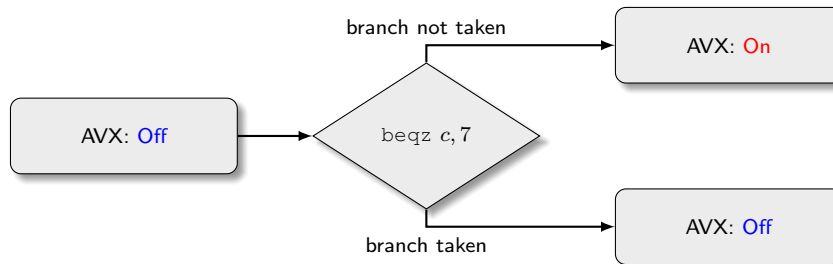
```

Listing 4.4: AVX Leak (C-Code)

```

1 c <- x < array1_size
2 beqz c, 7
3 load y, array1 + x
4 c <- y & 1
5 beqz c, 7
6 z <- a avx_add b

```

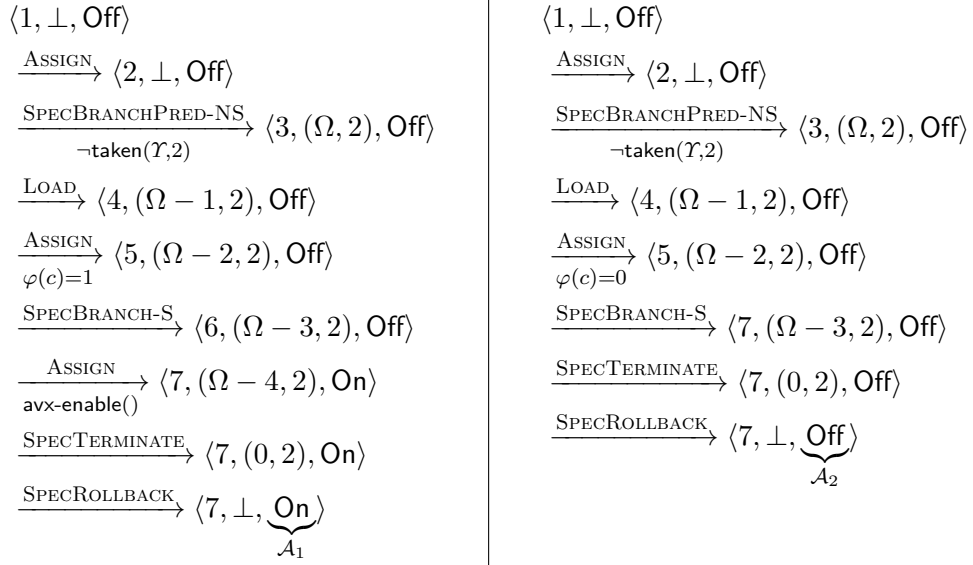
Listing 4.5: AVX Leak (μ ASM-Code)Figure 4.4: AVX Unit State Depending on the Least-Significant Bit of y

Listings 4.4 and 4.5 adopt the example from the introduction part of Chapter 4. The highlighted lines denote the implementation of `leak_least_significant_bit(y)`. We have $x = 10$ and `array1_size = 8`. Assume that the processor speculatively executes the code shown in Listing 4.5 and the AVX unit is **off** initially.

Because the adversary is in control of x , the load instruction at line 3 can speculatively load sensitive data into register y . The conditional jump at line 5 indirectly depends on the value of y , because the least-significant bit of y is assigned to register c at line 4. If c is 0 the jump is taken, causing that the vector instruction at line 6 is skipped. Otherwise, if c is 1 the jump is skipped but in this case the vector instruction at line 6 is executed, thereby enabling the AVX unit. Hence, depending on the value of c the AVX unit is either **on** or **off** at the end of execution. This way the least-significant bit of y can be encoded into the AVX unit. The two possible evolutions of the AVX unit are depicted in Figure 4.4.

Finally, the adversary can decode the information from the AVX unit by means of a timing attack. Measuring the execution time of a vector instruction reveals the least-significant bit of y . A long execution time, caused by first powering up the AVX unit, indicates that the AVX unit was **off**. This can only happen if the vector instruction at line 6 has been skipped, meaning that the least-significant bit of y was 0. A short execution time indicates that the AVX unit was already **on** when executing the vector instruction. This can only happen if the instruction at line 6 has been executed, meaning that the least-significant bit of y was 1.

Applying the AVX model to the example shown in Listing 4.5 gives two possible AVX unit states \mathcal{A}_1 and \mathcal{A}_2 , which are equivalent non-speculatively but differ speculatively. The simplified traces of both runs are shown in Figure 4.5.

Figure 4.5: AVX Unit Example Traces for $\text{LSB}(y) = 1$ and $\text{LSB}(y) = 0$

4.3 Branch Predictor

Modern processors make use of branch prediction to increase their overall performance [47]. Instead of waiting for the outcome of a branching decision, the processor speculatively executes instructions in advance by "predicting" their possible outcome. The more accurate the predictions are, the larger the possible performance gain is.

One integral part of this machinery is the branch predictor, which consists of two main components. One component is the Pattern History Table (PHT) which keeps track if branches have been taken/not-taken in the past [47]. The processor uses the information stored in the PHT to predict if future conditional branches should be taken or not. The second component is the Branch Target Buffer (BTB) which keeps track of the recently used jump target addresses [47]. The processor uses the information stored in the BTB to predict future target addresses when a branch is taken. Both components together allow the processor to speculatively execute conditional and unconditional branch instructions using direct or indirect branch targets [47].

Motivated by the Branch Prediction Analysis (BPA) attack [2, 3, 15], we model the BTB as well as the PHT for proving the absence of "secret-dependent" speculative execution flows. The BPA attack uses the BTB as a microarchitectural covert channel. A Prime+Probe like attack strategy allows to leak secret information used in branching decisions [3]. The adversary first trains the BTB with known target addresses, which map to the same BTB set as the victim branch. Then the victim function is executed, which may or may not overwrite the attacker's BTB entries depending on the branch outcome. Afterwards the adversary can check if the previously trained target addresses

have been evicted from the buffer. If so, the attacker knows that the victim branch has been taken.

4.3.1 PHT Model

A PHT state $\mathcal{P} \in \text{Word} \rightarrow \{\text{Taken}, \text{NotTaken}\} \cup \{\perp\}$ is a mapping from program locations to branching decisions, which can either be taken or not taken. In contrast to concrete implementations which are typically based on tagged data structures [47], our abstraction tracks the most recent branching decision for each individual program location instead of a set of program locations.

Initially the PHT contains an unknown branching decision for each program location.

$$\text{pht-init}() \equiv \{l \mapsto \perp \mid l \in \text{Word}\}$$

Storing a new branching decision for program location l just overwrites the current branching decision of l .

$$\begin{aligned} \text{pht-taken}(\mathcal{P}, l) &\equiv \mathcal{P}[l \mapsto \text{Taken}] \\ \text{pht-not-taken}(\mathcal{P}, l) &\equiv \mathcal{P}[l \mapsto \text{NotTaken}] \\ &\text{where } l \in \text{Word} \end{aligned}$$

Two PHT states $\mathcal{P}_1, \mathcal{P}_2$ are equivalent if both contain the same branching decisions for each program location.

$$\text{pht-equiv}(\mathcal{P}_1, \mathcal{P}_2) \equiv \forall l \in \text{Word}. \mathcal{P}_1(l) = \mathcal{P}_2(l)$$

4.3.2 PHT Configuration

The derivation relation $\xrightarrow{\text{pht-effects}}$ works on a configuration $\langle \mathcal{P}, \varphi, \sigma, pc, \varphi', \sigma', pc' \rangle$, where \mathcal{P} defines the current state of the PHT, φ the current register assignment, σ the current memory state, pc the current program counter, φ' the next register assignment, σ' the next memory state and pc' the next program counter.

4.3.3 PHT Instruction Evaluation

Correctly predicted branching decisions are immediately stored in the PHT, both for non-speculatively and speculatively executed branch instructions. Mis-predicted branching decisions are ignored until rollback. On rollback, when the mis-speculation is finally resolved, the correct branching decisions of the mis-speculated branch instruction is stored in the PHT. Intuitively, once the correct branch outcome is known, the correct branching decision can be stored. As the μarch semantics already implements this behavior, no special treatments is required for the `beqz` instruction and thus `PHTBRANCHTAKEN` and `PHTBRANCHNOTTAKEN` always update the PHT for the current program location pc . All other instructions leave the state of the PHT unaffected.

$$\text{PHTBRANCHTAKEN} \frac{\rho(pc) = \text{beqz } x, \ell \quad \varphi(x) = 0 \quad \mathcal{P}' = \text{pht-taken}(\mathcal{P}, pc)}{\langle \mathcal{P}, \varphi, \sigma, pc, \varphi', \sigma', pc' \rangle \xrightarrow{\text{pht-effects}} \mathcal{P}'}$$

$$\text{PHTBRANCHNOTTAKEN} \frac{\rho(pc) = \text{beqz } x, \ell \quad \varphi(x) \neq 0 \quad \mathcal{P}' = \text{pht-not-taken}(\mathcal{P}, pc)}{\langle \mathcal{P}, \varphi, \sigma, pc, \varphi', \sigma', pc' \rangle \xrightarrow{\text{pht-effects}} \mathcal{P}'}$$

$$\text{PHTIGNORE} \frac{\rho(pc) \neq \text{beqz } x, \ell}{\langle \mathcal{P}, \varphi, \sigma, pc, \varphi', \sigma', pc' \rangle \xrightarrow{\text{pht-effects}} \mathcal{P}}$$

4.3.4 BTB Model

A BTB state $\mathcal{B} \in \text{Word} \rightarrow \text{Word} \cup \{\perp\}$ is a mapping from program locations to target addresses. In contrast to concrete implementations which are typically based on tagged data structures [47], our abstraction tracks the most recently used target address for each individual program location instead of a set of program locations.

Initially the BTB contains an unknown target address for each program location.

$$\text{btb-init}() \equiv \{l \mapsto \perp \mid l \in \text{Word}\}$$

Tracking a target address t for program location l just overwrites the current target address of l .

$$\text{btb-track}(\mathcal{B}, l, t) \equiv \mathcal{B}[l \mapsto t] \quad \text{where } l, t \in \text{Word}$$

Two BTB states $\mathcal{B}_1, \mathcal{B}_2$ are equivalent if both contain the same target addresses for each program location.

$$\text{btb-equiv}(\mathcal{B}_1, \mathcal{B}_2) \equiv \forall l \in \text{Word}. \mathcal{B}_1(l) = \mathcal{B}_2(l)$$

4.3.5 BTB Configuration

The derivation relation $\xrightarrow{\text{btb-effects}}$ works on a configuration $\langle \mathcal{B}, \varphi, \sigma, pc, \varphi', \sigma', pc' \rangle$, where \mathcal{B} defines the current state of the BTB, φ the current register assignment, σ the current memory state, pc the current program counter, φ' the next register assignment, σ' the next memory state and pc' the next program counter.

4.3.6 BTB Instruction Evaluation

Correctly predicted branch targets are immediately stored in the BTB, both for non-speculatively and speculatively executed branch instructions. Mis-predicted branch targets are ignored until rollback. On rollback, when the mis-speculation is finally resolved, the correct branch target of the mis-speculated branch instruction is stored in the BTB. Intuitively, once the correct branch outcome is known, the correct branch target can be stored. As the μarch semantics already implements this behavior, no special treatments is required for the `beqz` instruction and thus `BTB` always tracks the branch target pc' for the current program location pc if the branch is taken. If the branch isn't taken, then the BTB is left unchanged by `BTB`. Additionally, the branch target pc' of unconditional branches `jmp` is tracked as well. All

other instructions leave the state of the BTB unaffected.

$$\begin{array}{l}
 \text{BTB_BRANCH_TAKEN} \frac{\rho(pc) = \text{beqz } x, \ell \quad \varphi(x) = 0 \quad \mathcal{B}' = \text{btb-track}(\mathcal{B}, pc, pc')}{\langle \mathcal{B}, \varphi, \sigma, pc, \varphi', \sigma', pc' \rangle \xrightarrow{\text{btb-effects}} \mathcal{B}'} \\
 \text{BTB_BRANCH_NOT_TAKEN} \frac{\rho(pc) = \text{beqz } x, \ell \quad \varphi(x) \neq 0}{\langle \mathcal{B}, \varphi, \sigma, pc, \varphi', \sigma', pc' \rangle \xrightarrow{\text{btb-effects}} \mathcal{B}} \\
 \text{BTB_JUMP} \frac{\rho(pc) = \text{jmp } \ell \quad \mathcal{B}' = \text{btb-track}(\mathcal{B}, pc, pc')}{\langle \mathcal{B}, \varphi, \sigma, pc, \varphi', \sigma', pc' \rangle \xrightarrow{\text{btb-effects}} \mathcal{B}'} \\
 \text{BTB_IGNORE} \frac{\rho(pc) \notin \{\text{beqz } x, \ell; \text{jmp } \ell\}}{\langle \mathcal{B}, \varphi, \sigma, pc, \varphi', \sigma', pc' \rangle \xrightarrow{\text{btb-effects}} \mathcal{B}}
 \end{array}$$

4.3.7 Example

Listings 4.6 and 4.7 adopt the example from the introduction part of Chapter 4. The highlighted lines denote the implementation of `leak_least_significant_bit(y)`. Given a BTB and a PHT with four slots each, the two lowest bits of the program location are used as the entry index. Hence, the conditional branch at line 2 has index `0b10` and the conditional branch at line 5 has index `0b01`. We have $x = 10$ and `array1_size = 8`. Assume that the processor speculatively executes the code shown in Listing 4.7 and the adversary trained the BTB to contain `0b01010101` at index `0b01` and the PHT to contain `Taken` at the same index.

```

1 if (x < array1_size) {
2     y = array1[x];
3     if (y & 1) {
4         _skip();
5     }
6 }

```

Listing 4.6: BTB/PHT Leak (C-Code)

```

1 c <- x < array1_size
2 beqz c, 7
3 load y, array1 + x
4 c <- y & 1
5 beqz c, 7
6 skip

```

Listing 4.7: BTB/PHT Leak (μ ASM-Code)

Because the adversary is in control of x , the load instruction at line 3 can speculatively load sensitive data into register y . The conditional jump at line 5 indirectly depends on the value of y , because the least-significant bit of y is assigned to register c at line 4. If c is 0 the jump is taken and the target address `0b00000111` is stored at index `0b01`, thereby overwriting the trained target address `0b01010101`. Otherwise if c is 1 the jump is skipped and the buffer is left unchanged. This way the least-significant bit of y can be encoded into the BTB. The two possible evolutions of the BTB are depicted in Figure 4.6. The additional target address `0b00000111` at index `0b10` comes from the conditional branch at line 2. The correct jump target of this branch is recorded after the rollback of the mis-speculation. The same applies to the PHT which contains different branching decisions depending on the least-significant bit of y , as shown in Figure 4.7.

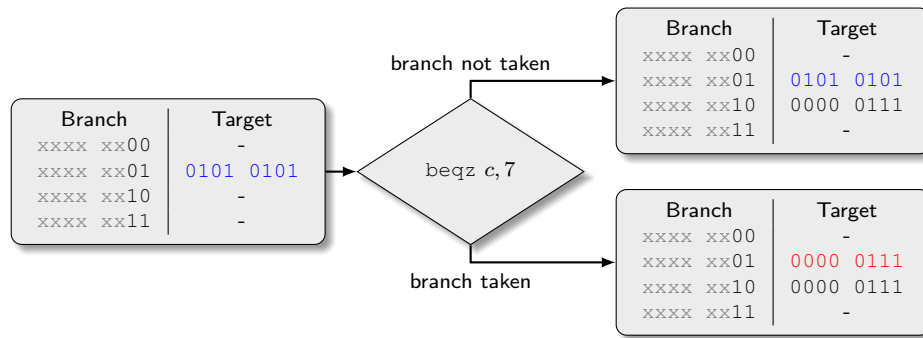


Figure 4.6: BTB State Depending on the Least-Significant Bit of y

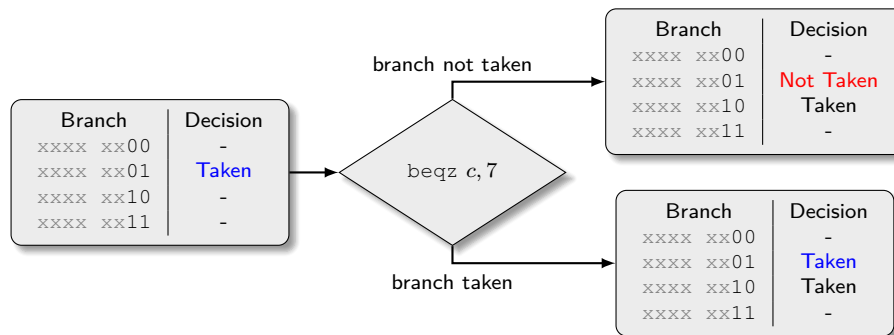
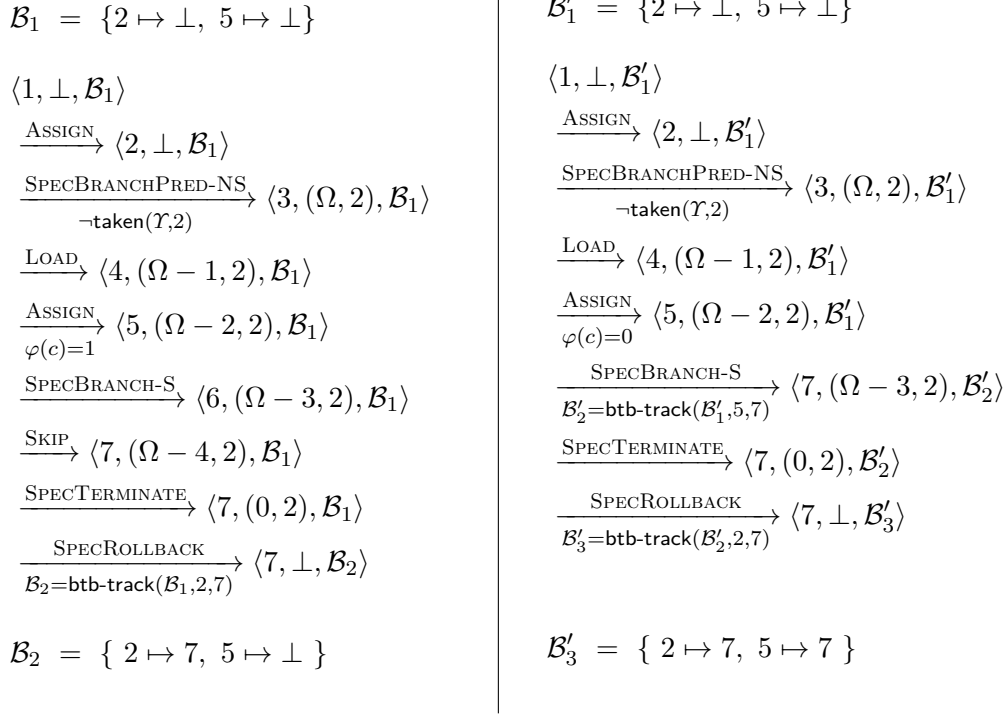


Figure 4.7: PHT State Depending on the Least-Significant Bit of y

Finally, the adversary can decode the information from the BTB and PHT by means of a timing attack. Measuring the execution time of a jump instruction, which maps to the same BTB slot $0b01$ as the victim’s branch instruction, reveals the least-significant bit of y . Assume that the correct target address of the specially crafted jump instruction is $0b01010101$. A long execution time indicates that the target address at index $0b01$ has been changed by the victim, thereby causing a mis-speculated jump to the buffered target address $0b00000111$. This can only happen if the branch at line 5 has been taken, meaning that the least-significant bit of y was 0. A short execution time indicates that the trained target address at index $0b01$ hasn’t been overwritten by the victim and therefore no costly mis-speculation is triggered. This can only happen if the branch at line 5 hasn’t been taken, meaning that the least-significant bit of y was 1.

Applying the BTB model to the example shown in Listing 4.7 gives two possible BTB states \mathcal{B}_2 and \mathcal{B}'_3 , which are equivalent non-speculatively but differ speculatively for the branch instruction at program location 5. The simplified traces of both runs are shown in Figure 4.8.

Figure 4.8: BTB Example Traces for $\text{LSB}(y) = 1$ and $\text{LSB}(y) = 0$

4.4 Combination of Microarchitectural Components

As the microarchitecture typically consist not only of one but instead of many microarchitectural components, we lift the component type \mathcal{K} as well as the associated functions defined in Section 3.4 to a tuple of component types $\mathcal{K}_1 \times \dots \times \mathcal{K}_n$. This enables us to simultaneously check multiple microarchitectural components in a single analysis, for example cache in combination with the branch target buffer.

$$\mathcal{K}_T \equiv \mathcal{K}_1 \times \dots \times \mathcal{K}_n$$

$$\kappa_T = (\kappa_1, \dots, \kappa_n) \in \mathcal{K}_T$$

$$\mathcal{K}_T\text{-init}() \equiv (\mathcal{K}_1\text{-init}(), \dots, \mathcal{K}_n\text{-init}())$$

$$\mathcal{K}_T\text{-equiv}(\kappa'_T, \kappa''_T) = \text{true} \text{ iff } \forall_{i=1}^n \mathcal{K}_i\text{-equiv}(\kappa'_i, \kappa''_i)$$

$$\frac{\kappa_T = (\kappa_1, \dots, \kappa_n) \quad \forall_{i=1}^n \kappa'_i = \mathcal{K}_i\text{-effects}(\kappa_i, \varphi, \sigma, pc, \varphi', \sigma', pc') \quad \kappa'_T = (\kappa'_1, \dots, \kappa'_n)}{\langle \kappa_T, \varphi, \sigma, pc, \varphi', \sigma', pc' \rangle \xrightarrow{\mathcal{K}_T\text{-effects}} \kappa'_T}$$



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Implementation

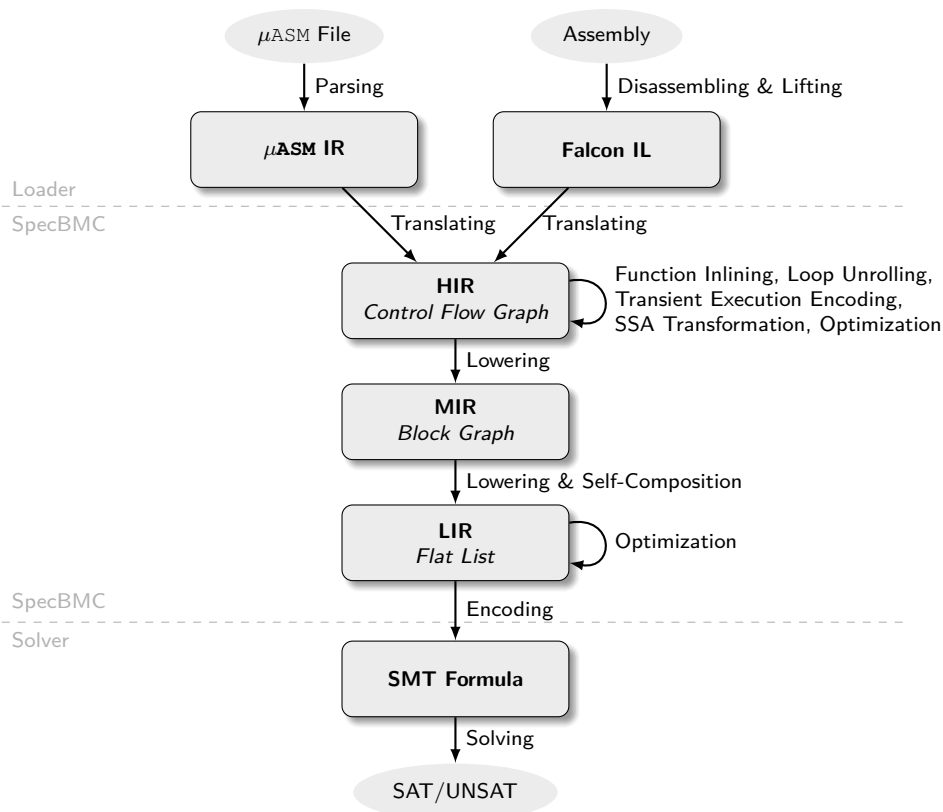


Figure 5.1: Architecture of SpecBMC

In this chapter the implementation and components of SpecBMC are explained¹. As

¹The full source code of SpecBMC is available at <https://github.com/emmanuel099/specbmc>

show in Figure 5.1, the architecture of SpecBMC can basically be divided into three distinct parts: (i) assembly loader (ii) the bounded model checker itself (iii) and an off-the-shelf satisfiability modulo theories (SMT) solver. In the first step the program, which can either be a μ ASM file or a binary program, is loaded and translated into a unified intermediate representation, called high-level intermediate representation (HIR). From there on a multitude of transformations and lowerings are performed to finally arrive at an SMT formula. The SMT formula is then passed on to an SMT solver which either returns unsatisfiable if the program is secure, or a satisfiable assignment (counterexample) if the program is insecure.

In the following all the different intermediate representations and involved transformation steps are described in more detail. For ease of understanding, all the explanations are based on the example program shown in Listing 5.1.

```

1  cond <- x < array1_size
2  beqz cond, EndIf
3  Then:
4  load v, array1 + x
5  load tmp, array2 + v << 8
6  EndIf:
7  skip

```

Listing 5.1: Kocher01 Example

5.1 Important Concepts

Before we dive into the implementation details of SpecBMC, we give a short recap of the two important concepts in use.

5.1.1 Bounded Software Model Checking

BMC is an automated approach for proving program properties [26]. The user supplies a program, some properties to be checked and an unwinding bound k . Then BMC does its job without further user interaction. To check the validity of the program in respect to the given properties, the program together with the properties is translated into a satisfiability problem, such that the problem is unsatisfiable if the program is invalid.

BMC tackles the state explosion problem by searching for counterexamples of bounded length [12]. Therefore, all loops in the program are unrolled k times, resulting in a loop free program where all execution paths are of finite length. The loop free program is then translated into a logic formula. If the formula is satisfiable, then a property has been violated. In this case the satisfiable assignment gives a counterexample to the validity of the program. If the formula is unsatisfiable, then the program respects all properties up to the given unwinding bound k . Note that a property can still be violated for unwinding bounds greater than k , meaning that BMC is incomplete if k is not sufficiently large. Nonetheless, BMC is known to be a strong technique for catching software bugs [26].

Roughly summarised, bounded software model checkers like LLBMC [34] or CBMC [11] work as follows: (i) inline function calls (ii) unwind the control-flow graph (CFG) up to k times to get a loop free CFG (iii) translate the CFG into static single assignment (SSA) form (iv) encode the SSA representation as SAT/SMT formula.

5.1.2 Self-Composition

Self-composition is a technique to reduce secure information flow problems into safety verification problems [6]. The reduction from the secure information flow policy of program P to a safety property of program P' is done by composing P with a duplicate of itself where variables are renamed, e.g., x to x^D . We have that P is secure if and only if P' is safe. The self-composition approach allows us to check our SNI 2-safety hyperproperty as defined in Section 3.5, over an execution of two copies of the program using standard safety verification techniques, such as BMC.

5.2 Loader

Currently two different loaders are implemented. One loader for assembly files based on Falcon and one loader for μ ASM files based on our μ ASM parser. In this section a short overview of both loaders is given.

5.2.1 Falcon

```
enum Operation {
  Assign { var: Variable, expr: Expression },
  Branch { target: Expression },
  Conditional { cond: Expression, operation: Operation },
  Store { address: Expression, expr: Expression },
  Load { var: Variable, address: Expression },
  Intrinsic { intrinsic: String },
  Nop,
}
```

Listing 5.2: Falcon Intermediate Language (IL)

Falcon² is a Binary Analysis Framework for 32/64-bit x86 and MIPS. The framework provides a multitude of ready to use components for implementing binary analysis solutions on top of it. The whole framework is built around the Falcon intermediate language (IL), which precisely captures the instruction semantics and control-flow. As noted by the authors of Falcon³: “Falcon IL is a simple, expression-based, well-defined, semantically-accurate intermediate language for the analysis of Binary Programs.”

The IL itself is surprisingly small and defines only seven operations as show in Listing 5.2. It consists of assignment, memory store and load as well as branching operations. The `Intrinsic` operation allows to capture instructions whose semantics cannot

²<https://github.com/falconre/falcon>

³<https://docs.rs/falcon/0.4.12/falcon/il/index.html>

be modelled by existing IL operations, e.g., speculation barriers. The `Conditional` operation allows to make each operation conditional, e.g., conditional assignments like `CMOVcc` or conditional branches like `Jcc`.

For instruction decoding Falcon relies upon the widely-used Capstone disassembly framework⁴. Each decoded instruction is lifted into one or more semantically equivalent Falcon IL instructions. The lifting itself is semantically accurate, meaning that all the stack, register as well as status flag⁵ modifications are captured. Instructions whose semantics cannot be modelled by Falcon IL are lifted as “intrinsic”. Such intrinsic operations allow us to find and insert speculation barriers when translating IL into HIR.

Translating Falcon IL into HIR

As Falcon IL already has a control-flow graph, the HIR control-flow graph is simply constructed from it. The IL operations itself are translated as follows:

- Unconditional assignments are translated as is.


```
IL::Assign{var, expr} ▶ HIR::Assign{var, expr}.
```
- Conditional assignments are translated into unconditional assignments, but instead the expression itself is made conditional. The new conditional expression is defined such that if the condition holds, the value of `expr` is assigned to `var`, otherwise the value of `var` is assigned to itself.


```
IL::Conditional{cond, IL::Assign{var, expr}} ▶
HIR::Assign{var, Ite(cond, expr, var)}
```
- The translation of unconditional branches depends on `target`. If the target is the address of a function as defined in the symbol table, then the branch operation is translated into a call instruction, otherwise the branch operation is translated into an unconditional branch instruction.


```
IL::Branch{target} ▶ HIR::Call{target} if target is a function
IL::Branch{target} ▶ HIR::Branch{target} otherwise
```
- Conditional branches are translated as is.


```
IL::Conditional{cond, IL::Branch{target}} ▶
HIR::ConditionalBranch{cond, target}
```
- Store and load operations are translated as is.


```
IL::Store{address, expr} ▶ HIR::Store{address, expr}
IL::Load{var, address} ▶ HIR::Load{var, address}
```
- Intrinsic operations are translated depending on the “raw” machine code instruction. Machine code instructions which behave as speculation barriers, like for

⁴<https://www.capstone-engine.org/>

⁵For example the carry, zero, sign and overflow flag in x86.

example “lfence”, are translated into barrier instructions. Others are simply translated into skip instructions.

IL::Intrinsic{intrinsic} ► HIR::Barrier if is a speculation barrier

IL::Intrinsic{intrinsic} ► HIR::Skip otherwise

- No operation is translated into a skip instruction.

IL::Nop ► HIR::Skip

5.2.2 μ ASM

For μ ASM files a parser has been written⁶. The input file is parsed and stored in the μ ASM intermediate representation (IR), basically just an abstract syntax tree (AST). The detailed syntax and semantics definition of μ ASM IR is given in Section 2.1.

Because HIR and μ ASM IR are closely related to each other, the translation into HIR is straightforward and therefore the detailed description of the translation process is omitted. The control-flow graph of HIR is constructed during translation. The resulting HIR program is shown in Figure 5.2.

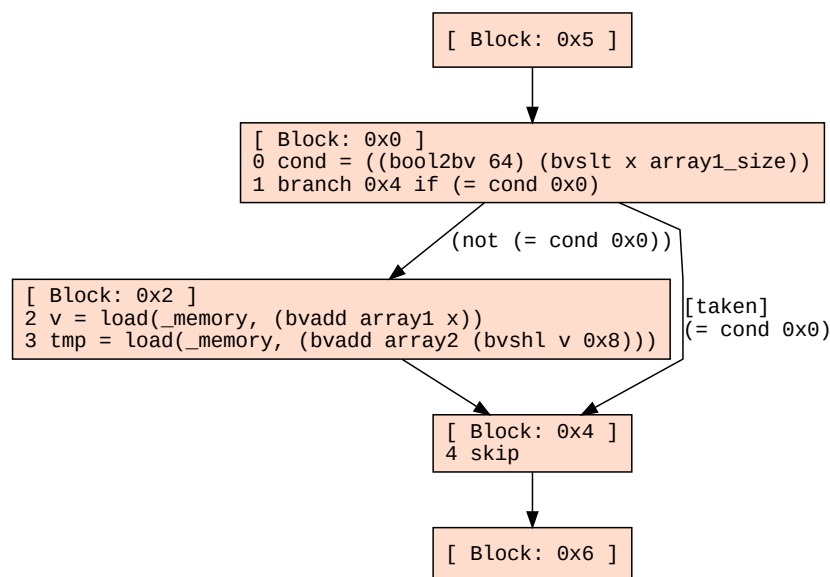


Figure 5.2: High-level Intermediate Representation of Kocher01 Example

5.3 SpecBMC

SpecBMC is divided into three intermediate representations called high-, mid- and low-level intermediate representation, in the following denoted as HIR, MIR and LIR. Following the principle of “separation of concerns”, each IR is optimized for a specific level

⁶ μ ASM Parser: <https://github.com/emmanuel099/muasm-parser>

of abstraction. While HIR has detailed knowledge about speculative execution, memory operations and such, the more low-level the IR becomes, the more such details are abstracted away. This more general representation of MIR and LIR allows to easily reuse them for other similar problems as well, whereas HIR is closely tight to the problem of this thesis.

Shared among all three intermediate representation of SpecBMC is an extensive expression library which encapsulates all the necessary SMT-LIB theories⁷ in a type-safe way. At the moment the theories of arrays, bit vectors and integers are available. Additionally, a type-safe abstraction for lists, tuples, byte-based memory and all the microarchitectural components, as defined in Chapter 4, is provided.

SpecBMC can fully be configured via an environment file. A full reference of the environment file is given in Appendix B.2. Whenever we use the term environment in the following, we actually refer to this environment file.

5.3.1 High-level Intermediate Representation (HIR)

```
enum Instruction {
  Assign { var: Variable, expr: Expression },
  Store { addr: Expression, expr: Expression,
         mem_in: Variable, mem_out: Variable },
  Load { var: Variable, addr: Expression, mem: Variable },
  Call { target: Expression },
  Branch { target: Expression },
  ConditionalBranch { cond: Expression, target: Expression },
  Skip,
  Barrier,
  Assert { cond: Expression },
  Assume { cond: Expression },
  Observable { expr: Expression },
  Indistinguishable { expr: Expression },
}

struct PhiNode {
  incoming: Map<BlockRef, Variable>,
  out: Variable,
}

struct Block {
  instructions: Vec<Node>,
  phi_nodes: Vec<PhiNode>,
}

struct Edge {
  condition: Option<Expression>,
}
```

⁷<http://smtlib.cs.uiowa.edu/theories.shtml>


```

struct Function {
    control_flow_graph: DirectedGraph<Block, Edge>,
}

struct Program {
    functions: Vec<Function>,
}

```

Listing 5.3: High-level Intermediate Representation

Listing 5.3 shows the basic structure of HIR. A program consists of one or multiple functions. Each function is defined by its control-flow graph (CFG), a directed graph of basic blocks and control-flow edges. An edge can either be conditional, meaning that the edge is taken only when the condition holds, or unconditional, meaning that the edge is always taken. Each basic block holds a linear sequence of instructions and a list of phi nodes. A phi node merges variables from different incoming control-flows, meaning that the variable `out` is selected to be one of the incoming variables, depending on from which predecessor basic block the control-flow is coming from [42]. Phi nodes are initially empty until HIR is transformed into static single assignment (SSA) form. The instructions of HIR closely follow the definition of Chapter 2. The only difference is that HIR allows to model function calls, which becomes a necessity when checking larger binary programs, as such programs are usually built of multiple functions. But as we shall see later, calls are vanished once function inlining has been performed.

HIR uses a flat memory model with a single continuous address space, meaning that each single byte in the memory is individually addressable. Flat memory models are already successfully used in other model checkers, like for example LLBMC [48]. Additionally, the state of the memory is explicitly tracked for each memory operation. This is different from typical IR implementations, where memory operations usually work on an implicit memory state.

The instructions of HIR are as follows:

- `Assign{var, expr}` denotes an assignment of expression `expr` to variable `var`.
- `Store{addr, expr, mem_in, mem_out}` stores the value of expression `expr` at the memory address `addr`. The store operation uses the memory state of variable `mem_in` as input and results in a new memory state which is assigned to variable `mem_out`.
- `Load{var, addr, mem}` loads the value at the memory address `addr` into variable `var`. The load operation uses the memory state of variable `mem`.
- `Call{target}` denotes a function call, where `target` is the address of the function to be called.
- `Branch{target}` and `ConditionalBranch{cond, target}` denote an unconditional respectively conditional jump to program location `target`. Note that the semantics of direct branches is already reflected by control-flow edges and thus

the branch instructions may seem to be partially redundant in HIR. The reason for retaining direct branches is solely to keep track of their side-effects.

- `Barrier` denotes a speculation barrier.
- `Assert(cond)` assertions that condition `cond` is true.
- `Assume(cond)` assumes that condition `cond` is true.
- `Observable(expr)` denotes that the value of expression `expr` is visible to an adversary. The observable instruction ensures that the expression has identical values in all compositions. If the values are different, meaning that high-security information has been leaked into `expr`, the program will be marked as insecure.
- `Indistinguishable(expr)` denotes that the value of expression `expr` is indistinguishable to an adversary. The indistinguishable instruction forces that the expression has identical values in all compositions. This is useful, e.g., to initialize low-security input variables and memory locations to identical values.

In the following the different HIR transformations are described. The first transformation is function inlining, which transforms a program with multiple functions into a single CFG. Then loops are unrolled, transient execution behavior is encoded and observations are added. Finally, the CFG is transformed into SSA form.

Function Inlining

Function inlining replaces call instructions with a copy of the control-flow graph of the target function. If a function f is called multiple times, then each call is replaced by a new copy of f to preserve context-sensitivity. For recursive functions the number of inlinings is limited by the *recursion limit* parameter, meaning that once the limit for a recursive function g is reached, no further inlining of function g is performed. In SpecBMC function inlining starts from the program entry function and repeatedly inlines function calls until all calls have been processed. The result is a program with a single CFG where all calls have been eliminated, meaning that all further transformations and translations don't have to deal with functions and call instructions anymore.

Figure 5.3 shows an example for function inlining. We have three functions A , B and C where A calls B and B calls C . The corresponding call graph is depicted in Figure 5.3d. Suppose that function A is the program entry function, then function inlining will produce the CFG as shown in Figure 5.3e.

Loop Unrolling

Loop unrolling, also called loop unwinding, is an integral part of BMC [26]. Loops are unrolled up to a given unwinding bound k by replacing them with a sequence of k copies of their loop bodies. To avoid that more than k iterations are possible, an unwinding assumption is added to the last iteration of each loop, making sure that the loop condition is `false` at the end of the k -th iteration. Furthermore, control-flow edges are inserted such that loops can be left before or after each iteration, making sure that

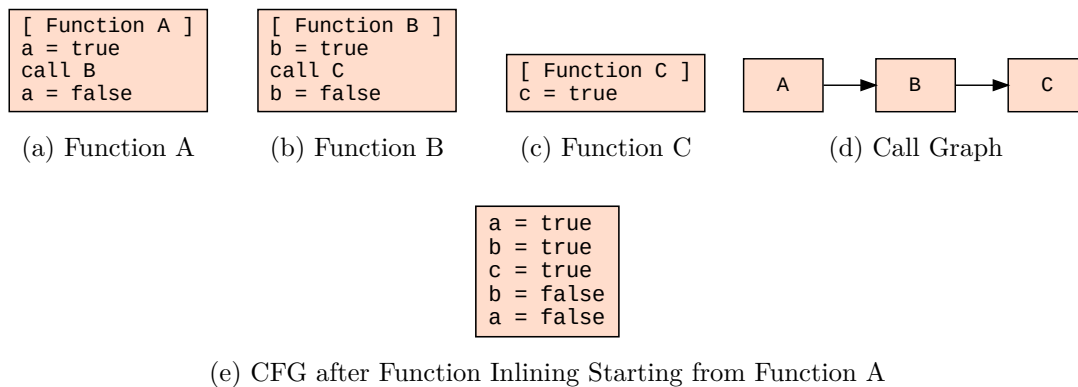


Figure 5.3: Function Inlining Example

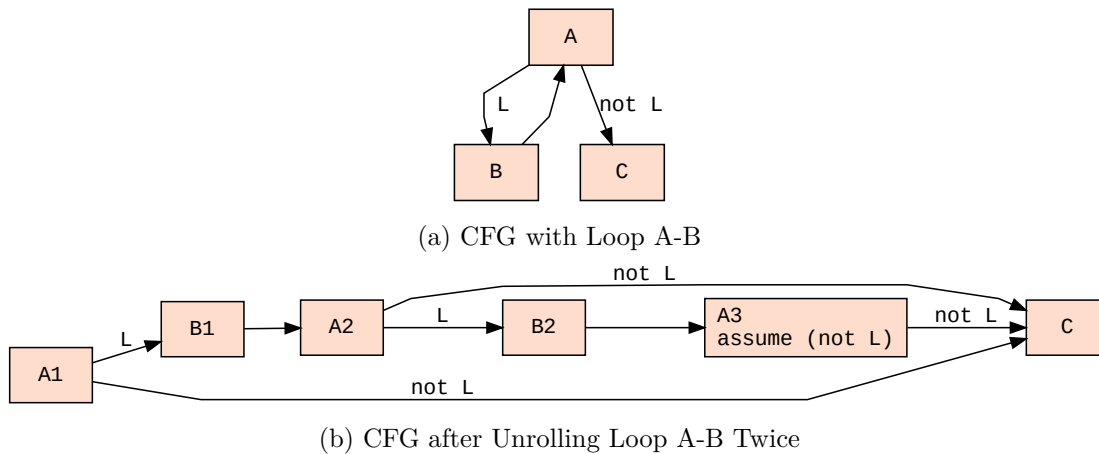


Figure 5.4: Loop Unrolling Example

executing only 0, 1 up to k loop iterations is possible. By inserting unwinding assertions instead of assumptions, the check would fail if for one loop the given bound k would be exceeded. Therefore, unwinding assertions allow to prove that the chosen bound is indeed large enough. SpecBMC allows the user to chose between unwinding assumptions and unwinding assertions. The default is unwinding assumptions. After loop unrolling the CFG forms a directed acyclic graph (DAG). During this transformation, all execution paths which were previously of infinite length became finite/bounded.

An example for loop unrolling is shown in Figure 5.4. Assume that k is two, meaning that the loop A-B is unrolled twice. A1-B1 is the copy for the first loop iteration and A2-B2 is the copy for the second loop iteration. As the unwinding bound is limited to two, no more loop iterations are expected and therefore an unwinding assumption with the negated loop condition (`not L`) is added to block A3.

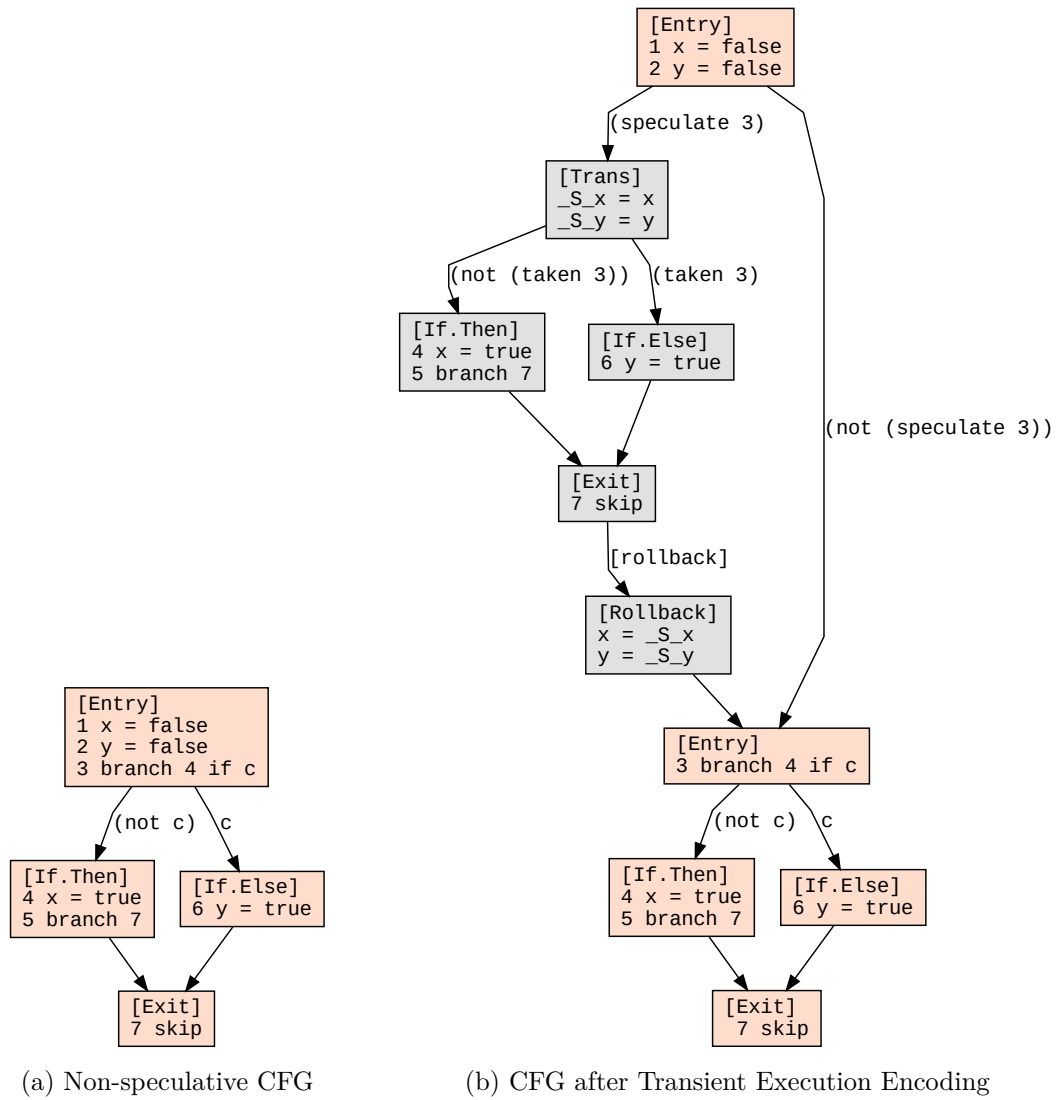


Figure 5.5: Transient Execution Transformation Example

Transient Execution

Up until now the CFG was non-speculative. In this transformation the transient execution semantics as described in Section 3.3 is encoded into the CFG. This is done by duplicating basic blocks and adding additional speculate, predict and rollback edges between them. The additional basic blocks and edges are added such that all execution paths given by our transient execution semantics are represented in the CFG. Furthermore, save and restore instructions for modified registers and memory are added, mimicking the behavior of the speculation state $\Delta = (\omega, \hat{\varphi}, \hat{\sigma}, \widehat{pc})$ from our semantics definition.

Figure 5.5 shows a simplified example for the transient execution encoding. The given non-speculative CFG is shown in Figure 5.5a. As the branch instruction in the entry block at program location 3 can be executed speculatively, the transient execution behavior for this instruction is inserted into the CFG as depicted in Figure 5.5b. The grey basic blocks denote the transiently executed blocks. First, the entry block is split into two blocks such that all instructions before the branch instruction are executed independently from the speculation. Then the blocks reachable from the branch instruction, namely `If.Then`, `If.Else` and `Exit`, are duplicated. The depth of the duplicated sub-graph is statically limited by the given maximum length of the speculation window, denoted as Ω in our semantics definition. As the control-flow during speculation is given by the predictor, (`taken 3`) respectively (`not (taken 3)`) edges are added, replacing the `c` respectively (`not c`) edges from the non-speculative execution. From the last block of the transient execution, a rollback edge is added such that the branch instruction is re-evaluated after transient execution. The saving and restoring of the modified variables x and y is done in `Trans` and `Rollback`. For simplicity all the additional resolve edges have been omitted from the example.

Observations

Observations are automatically inserted by SpecBMC depending on the specified observation model. Note that we omit the abstract `obs` instruction of the transient execution semantics and instead encode the observations directly into HIR, by making the relevant expressions and variables observable respectively indistinguishable to an adversary. Currently three different observation models are implemented:

- **Trace:** The trace observation model describes an adversary who can observe the state of the microarchitectural components while running in parallel with the victim. The implementation of this model follows the definition in Section 3.4. For each instruction with side effects on microarchitectural components, we trace the microarchitectural state after the instruction's effects have been encoded. The trace is implemented as a list of component states. At the end of the program the traces are compared. The program is secure if the traces of both executions are identical. When we relate the implementation of the trace observation model to the SNI hyperproperty described in Section 3.5, then `@observe(trace)` corresponds to $\tilde{\pi}_1 \approx_{\text{Obs}} \tilde{\pi}_2$ and the trace itself corresponds to `observations(π^μ)` $\in \mathcal{K}^*$.

<pre> 1 trace = nil 2 c <- x < array1_size 3 @track-pht(c) 4 @track-btb(EndIf) if c=0 5 trace = trace < <cache,pht,btb> 6 beqz c, EndIf 7 Then: 8 @cache-fetch(array1 + x) 9 trace = trace < <cache,pht,btb> 10 load v, array1 + x 11 @cache-fetch(array2 + v) 12 trace = trace < <cache,pht,btb> 13 load tmp, array2 + v 14 EndIf: 15 skip 16 observable(trace) </pre>	<pre> 1 c <- x < array1_size 2 c <- x < array1_size 3 4 @track-pht(c) 5 @track-btb(EndIf) if c=0 6 beqz c, EndIf 7 Then: 8 9 @cache-fetch(array1 + x) 10 load v, array1 + x 11 12 @cache-fetch(array2 + v) 13 load tmp, array2 + v 14 EndIf: 15 skip 16 observable(cache, pht, btb) </pre>
--	---

(a) Trace Observation Model

(b) Sequential Observation Model

```

1  c <- x < array1_size
2  @track-pht(c)
3  @track-btb(EndIf) if c=0
4  observable(cache, pht, btb)
5  beqz c, EndIf
6  Then:
7  @cache-fetch(array1 + x)
8  observable(cache, pht, btb)
9  load v, array1 + x
10 @cache-fetch(array2 + v)
11 observable(cache, pht, btb)
12 load tmp, array2 + v
13 EndIf:
14 observable(cache, pht, btb)
15 skip

```

(c) Parallel Observation Model

Figure 5.6: Observation Models applied to Kocher01 Example

Figure 5.6a shows the application of the trace observation model to the Kocher01 example. Initially the trace is empty as indicated by `nil`. Then on each side effect, denoted by `@`, the components' current state is appended to the trace. Finally at line 16 the traces of both executions are compared.

- **Sequential:** The sequential observation model describes an adversary who can observe the state of the microarchitectural components only once after the victim code has been executed. The program is secure if the resulting microarchitectural states of both executions are identical. The sequential observation model is basi-

cally a simplified version of the trace observation model, with a single element in the trace. Therefore, we omit the list representation and instead directly compare the microarchitectural states at the end of the program.

Figure 5.6b shows the application of the sequential observation model to the Kocher01 example. The effects of the branch and load instructions are encoded into the component states. Finally at line 16 the component states are compared.

- **Parallel:** The parallel observation model describes an adversary who can observe the state of the microarchitectural components while running in parallel with the victim. The parallel model is a simplification of the trace model, thereby sacrificing analysis soundness in favor of a simpler and potentially faster to solve SMT formula. Instead of collecting the microarchitectural states in a trace, we let an adversary directly observe the microarchitectural states, thereby avoiding the need of the theory of lists and tuples⁸. The direct comparison of the component states has of course the implication, that the comparison can only be done if the observation is actually performed in both executions. Roughly speaking, if we take a full trace as generated by the trace observation model and only keep the pairs of elements where both execution paths overlap, then we basically have the parallel observation model. In the parallel model observations are added to all instructions with side effects and additionally to each control-flow join.

The restriction of the parallel observation model has of course impacts on the leaks which can be caught respectively which are missed. Suppose that we have a set of side effects E_1 and E_2 . Then the parallel observation model allows to catch the following leaks:

$$\text{if (} \underbrace{\text{cond}}_{\text{control leak if secret}} \text{) } \left\{ \underbrace{\text{permutation}(E_1)}_{\text{data leak if contains secret}} \right\} \text{ else } \left\{ \underbrace{\text{permutation}(E_2)}_{\text{data leak if contains secret}} \right\}$$

control leak if $(E_1 \setminus E_2) \cup (E_2 \setminus E_1) \neq \emptyset$

The observation of the condition depends of course on the available components. Suppose that only the cache is available, then control flow leaks can be detected if the set of side effects of both branches differ, meaning that the symmetric difference of E_1 and E_2 is not empty. The parallel observation model misses control flow leaks if the set of effects E_1 and E_2 are equal, even if the effects appear in different order in both branches. Meaning that the parallel model is less powerful than the trace model, which allows to catch control flow leaks if the same set of effects appear in different order. An example for this is `@cache-fetch(1); @cache-fetch(2)` in the then branch and `@cache-fetch(2); @cache-fetch(1)` in the else branch.

Figure 5.6c shows the application of the parallel observation model to the Kocher01 example. The side effects of the branch and load instructions are directly compared after they have been encoded into the component's state.

⁸SMT formulas generated with the parallel observation model only require the QF_AUFBV logic.

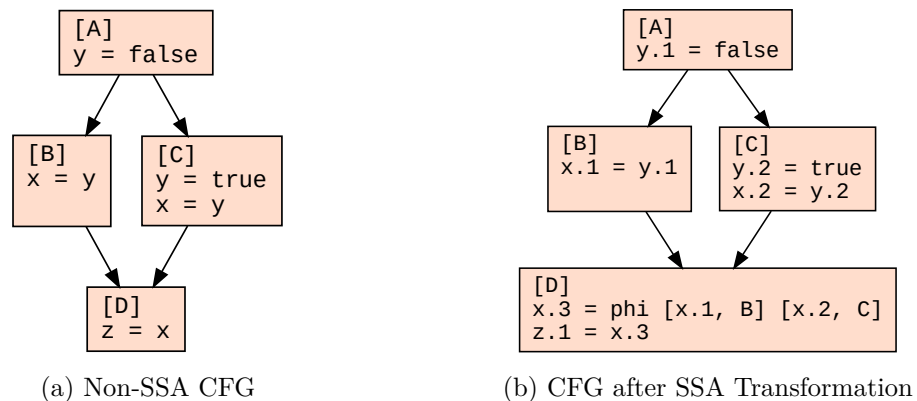


Figure 5.7: SSA Transformation Example

In addition to the observations, we add indistinguishable constraints to HIR such that the non-speculative part of the program is observational equivalent to an adversary as required by $\text{ns}(\tilde{\pi}_1) \approx_{\text{Obs}} \text{ns}(\tilde{\pi}_2)$. To achieve this we add non-speculative counterparts for all microarchitectural components, denoted by the `_ns` suffix, which only incorporate side effects of non-speculatively executed instructions, and force them to be indistinguishable.

In the following we make use of the sequential observation model.

Single Static Assignment (SSA) Transformation

The authors of SSA-based Compiler Design [42] informally define SSA form as follows: “A program is defined to be in SSA form if each variable is a target of exactly one assignment statement in the program text.” Adapted to HIR this means, that a program is defined to be in SSA form if each variable is a target of exactly one assign, load or store instruction.

A program with multiple assignments for the same variable can be transformed into single-assignment form by means of variable renaming [42], thereby a new unique variable is introduced for each assignment. Take the assignments to variable `y` in block A and B as shown in Figure 5.7a as an example. The assignments are transformed into SSA form by renaming the target variable `y` to `y.1` respectively `y.2`, as shown in Figure 5.7b. The process of variable renaming is repeated for each variable until all variables have been renamed. As variables may be used (read) in other instructions, all the variable uses have to be adjusted as well. Each use of a variable `v` is replaced by the latest prior SSA definition of variable `v`. In our example the variable `y` is used twice, once in block B and once in block C. In block B the latest prior definition of `y` is `y.1`, meaning that the variable `y` in the assignment to variable `x.1` is replaced by `y.1`. In block C the latest prior definition of `y` is `y.2`, meaning that the variable `y` in the assignment to variable `x.2` is replaced by `y.2`.

Special care has to be taken at control flow joins, such as block D in Figure 5.7a. Depending on from where the control-flow is coming from the value of `x` is different. But

in SSA form we have $x.1$ and $x.2$ instead of x , so which of them should be assigned to z ? – It depends on the control-flow. In SSA this choice is reflected by so-called phi nodes, usually referred to as ϕ functions. A phi node merges variables from different control-flows into a single variable, meaning the outgoing variable is selected to be one of the incoming variables depending on from which predecessor basic block the control-flow is coming from [42]. As shown in Figure 5.7b, the variables $x.1$ and $x.2$ are merged into variable $x.3$, such that $x.3$ is selected to be $x.1$ if the control-flow is coming from block B or $x.2$ if the control-flow is coming from block C.

SpecBMC implements the “pruned” SSA transformation algorithm [42]. The pruned version inserts phi nodes only where absolutely necessary. The algorithm relies on live variable analysis [4] to determine where phi nodes are required. In general the pruning is rather expensive because of the live variable analysis and therefore often avoided in favor of the less expensive semi-pruned SSA transformation [42]. But as all loops in HIR are eliminated by means of loop unrolling, the analysis operates on DAGs only. Therefore, the expensive worklist-based fixed-point computation required for the live variable analysis can be replaced by a simple reversed top-sort ordering traversal, making the live variable analysis almost as cheap as the global variable analysis required for the semi-pruned version. For example, in Figure 5.7a only variable x is live at the entry of block D, meaning that in block D only a phi node for x needs to be added, whereas the phi node for y can be omitted. The semi-pruned SSA transformation would also add a phi node for y , even if y is not used in D.

Optimization

After the HIR is in SSA form, some simple optimizations are performed. These optimizations allow us for example to completely get rid of the additional save and restore instructions introduced by the transient execution transformation. Currently the following optimizations are implemented:

- **Constant Propagation:** Propagates all constant assignments but doesn’t remove them. Constant assignments are defined as $x = c$ where c is a constant. For example, suppose that we have $x = c$ and $y = \text{load}(x)$, then constant propagation will give $x = c$ and $y = \text{load}(c)$. Constants are propagated to instructions and control-flow edges.
- **Copy Propagation:** Propagates all simple assignments but doesn’t remove them. Simple assignments are defined as $x = v$ where v is a variable. For example, suppose that we have $x = v$ and $y = \text{load}(x)$, then constant propagation will give $x = v$ and $y = \text{load}(v)$. Copies are propagated to instructions, control-flow edges and phi nodes.
- **Expression Simplification:** Tries to simplify expressions, e.g., $x \wedge \text{false}$ will become false . Expression simplification is done for all instructions and control-flow edges. In total, approximately 50 simplification rules have been implemented.

- **Constant Folding:** Tries to evaluate expressions to constants if all their operands are constant, e.g. $1 + 2$ will become 3. Constant folding is done for all instructions and control-flow edges. In total, approximately 20 folding rules have been implemented.
- **Dead Code Elimination:** Removes unused variable assignments and phi nodes. Implemented as a mark-and-sweep algorithm which in the first phase marks all unused assignments and phi nodes, and in the second phase removes them.
- **Phi Node Elimination:** Replaces unnecessary phi nodes with assignments. Copy propagation may produce phi nodes where all incoming variables are the same, i.e., $x' = \text{phi } [x_1, b_1] [x_1, b_2] \dots [x_1, b_n]$. Such phi nodes are transformed into $x' = x_1$.
- **Redundant Instruction Elimination:** First computes the set of available instructions for each block and then removes all instructions which are already available at their position. This optimization is useful to get rid of redundant assertions and assumptions.

When applying all the described transformations and optimization to the input program shown in Figure 5.2, we get the CFG shown in Figure 5.8 as a result. This CFG is then lowered into MIR as described in the following section.

5.3.2 Mid-level Intermediate Representation (MIR)

```

enum Node {
  Let { var: Variable, expr: Expression },
  Assert { cond: Expression },
  Assume { cond: Expression },
  HyperAssert { cond: Expression },
  HyperAssume { cond: Expression },
}

struct Block {
  execution_condition: Expression,
  nodes: Vec<Node>,
}

struct Program {
  block_graph: Graph<Block>,
}

```

Listing 5.4: Mid-level Intermediate Representation

The design of MIR has greatly been inspired by the ideas of LLBMC's ILR [34], namely the concept of execution conditions and phi node transformation.

As shown in Listing 5.4, a MIR program consists of a block graph. Each block is a list of variable bindings, assertions and assumptions together with an execution condition. The execution condition is basically a guard which tells if the block is executed or not. The difference between assertions/assumptions and hyper-assertions/hyper-assumptions

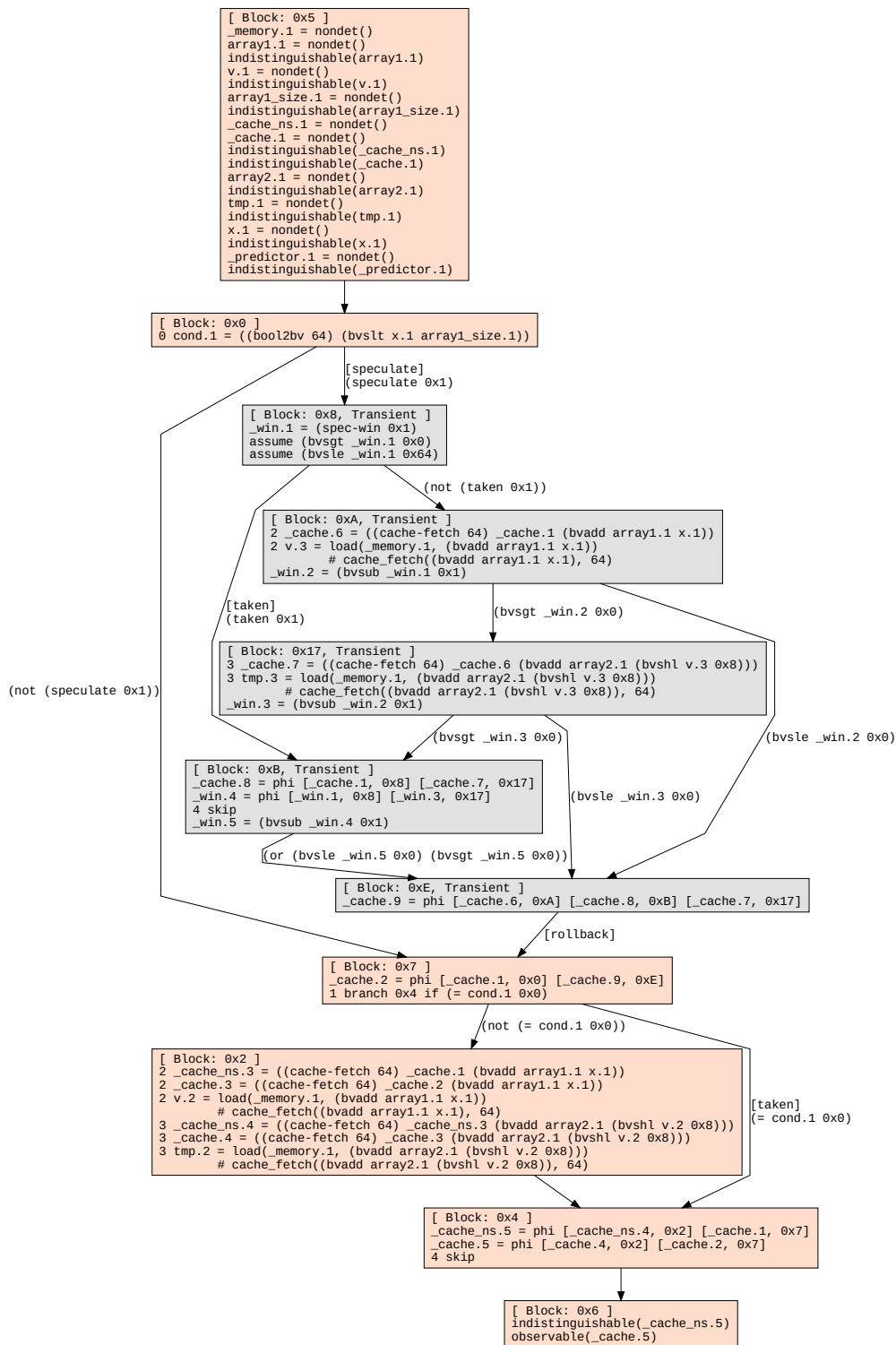


Figure 5.8: Transformed High-level Intermediate Representation of Kocher01 Example

is, that the former describe properties while the latter describe hyperproperties. Therefore, conditions of assertions/assumptions only refer to variables of the same composition while conditions of hyper-assertions/hyper-assumptions refer to variables of different compositions.

Note that MIR uses a graph structure only for ease of debugging, therefore the edges of the block graph have no meaning beyond visually indicating the relationship among blocks retained from the higher level. Technically this wouldn't be necessary because the execution condition makes the blocks order-independent.

Lowering (HIR \rightarrow MIR)

Lowering of HIR into MIR consists of three parts: (i) computing the execution condition of each block (ii) transforming phi nodes into conditional expressions and (iii) translating HIR instructions into MIR nodes.

First, the execution condition $\text{exec}(b)$ is computed for each basic block. Let the execution condition $\text{exec}(b)$ of basic block b be a predicate such that it evaluates to `true` if either b itself is the CFG entry (no predecessors), or the execution condition of an immediate predecessor block p of b is `true` and the edge from p to b is taken. More formally: Let $\text{pred}(b)$ denote the set of immediate predecessors of basic block b in the CFG. Additionally, let $t(p, b)$ be the condition of the edge from block p to b . For unconditional edges $t(p, b)$ is defined as `true`. The execution condition of a basic block b is then computed as follows [34]:

$$\text{exec}(b) = \begin{cases} \text{true} & \text{if } \text{pred}(b) \text{ is empty} \\ \bigvee_{p \in \text{pred}(b)}. (\text{exec}(p) \wedge t(p, b)) & \text{otherwise} \end{cases}$$

In the next step all phi nodes are transformed into conditional expressions similar as done in LLBMC [34]. Let $x' = \text{phi } [x_1, b_1] [x_2, b_2] \dots [x_n, b_n]$ denote a generalized form of a phi node as it appears in HIR. Additionally, let b' denote the basic block where the phi node is located. The meaning of the phi node is, that x' is chosen to be x_i if the control-flow is coming from the predecessor block b_i . By making use of the previously defined execution condition, the implicit control-flow based on basic blocks can be made explicit, meaning that x' is chosen to be x_i if $\text{exec}(b_i) \wedge t(b_i, b')$ holds. Hence, phi nodes can be lowered into a sequence of conditional expressions as follows [34]:

$$\begin{aligned} x' &= \text{phi } [x_1, b_1] [x_2, b_2] \dots [x_n, b_n] \blacktriangleright \\ \text{MIR}::\text{Let}\{ &x', \text{Ite}(\text{exec}(b_1) \wedge t(b_1, b'), x_1, \\ &\text{Ite}(\text{exec}(b_2) \wedge t(b_2, b'), x_2, \\ &\text{Ite}(\dots, \text{Ite}(\text{exec}(b_{n-1}) \wedge t(b_{n-1}, b'), x_{n-1}, x_n) \dots))\} \end{aligned}$$

Finally, the HIR instructions of each basic block are lowered as follows:

- The SSA-form of HIR guarantees that each variable is only assigned once [42]. Therefore, assignments can simply be lowered as variable bindings.

$$\text{HIR}::\text{Assign}\{\text{var}, \text{expr}\} \blacktriangleright \text{MIR}::\text{Let}\{\text{var}, \text{expr}\}$$

- Assertions and assumptions are translated as is.

```
HIR::Assert{cond} ▶ MIR::Assert{cond} respectively
HIR::Assume{cond} ▶ MIR::Assume{cond}
```

- As the memory state is already explicitly tracked as variable for all `Store` instructions, memory stores can be lowered as variable bindings and thereby transforming them into store expressions.

```
HIR::Store{addr, expr, mem_in, mem_out} ▶
MIR::Let{mem_out, ((store «bitwidth») mem_in addr expr)}
```

As indicated by `«bitwidth»`, the store operator depends on the actual bit-width of the value to be stored. For example, if `expr` is of type `BitVec<64>`, the resulting store operator is `(store 64)`.

- Memory loads are lowered similar to memory stores. For memory loads the bit-width is given by the type of the target variable `var`.

```
HIR::Load{var, addr, mem} ▶
MIR::Let{var, ((load «bitwidth») mem addr)}
```

- The `Indistinguishable` instructions causes that the *expression* is indistinguishable to an adversary. Therefore, such instructions are lowered as an hyper-assumption which assumes that the *expression* is equal under self-composition.

```
HIR::Indistinguishable{expr} ▶
MIR::HyperAssume{expr@0 = expr@1}
```

- The *expression* of an `Observable` instruction is visible to an adversary. As leaks should be avoided, it must be assured that the *expression* doesn't actually contain secret information. Therefore, such instructions are lowered as an hyper-assertion which asserts that the *expression* is equal under self-composition.

```
HIR::Observable{expr} ▶ MIR::HyperAssert{expr@0 = expr@1}
```

- Other instructions, namely `Skip`, `Call`, `Branch`, `ConditionalBranch` and `Barrier`, don't need to be explicitly lowered as they are already implicitly encoded into the CFG by the various different HIR transformations.

Figure 5.9 shows the Kocher01 example lowered to MIR. The execution condition of each block is given in the squared brackets beside the block ID. An example for an lowered phi node can be found at line 1 of block `0x4`.

5.3.3 Low-level Intermediate Representation (LIR)

LIR is designed to be simple and close to SMT while still being solver-agnostic. This allows us to easily target a wide range of different SMT solvers, even using their own proprietary API instead of the SMT-LIB⁹ interface, without duplicating all the lowering and self-composition work. As shown in Listing 5.5, LIR only consists of variable

⁹SMT-LIB Language: <http://smtlib.cs.uiowa.edu/language.shtml>

5. IMPLEMENTATION

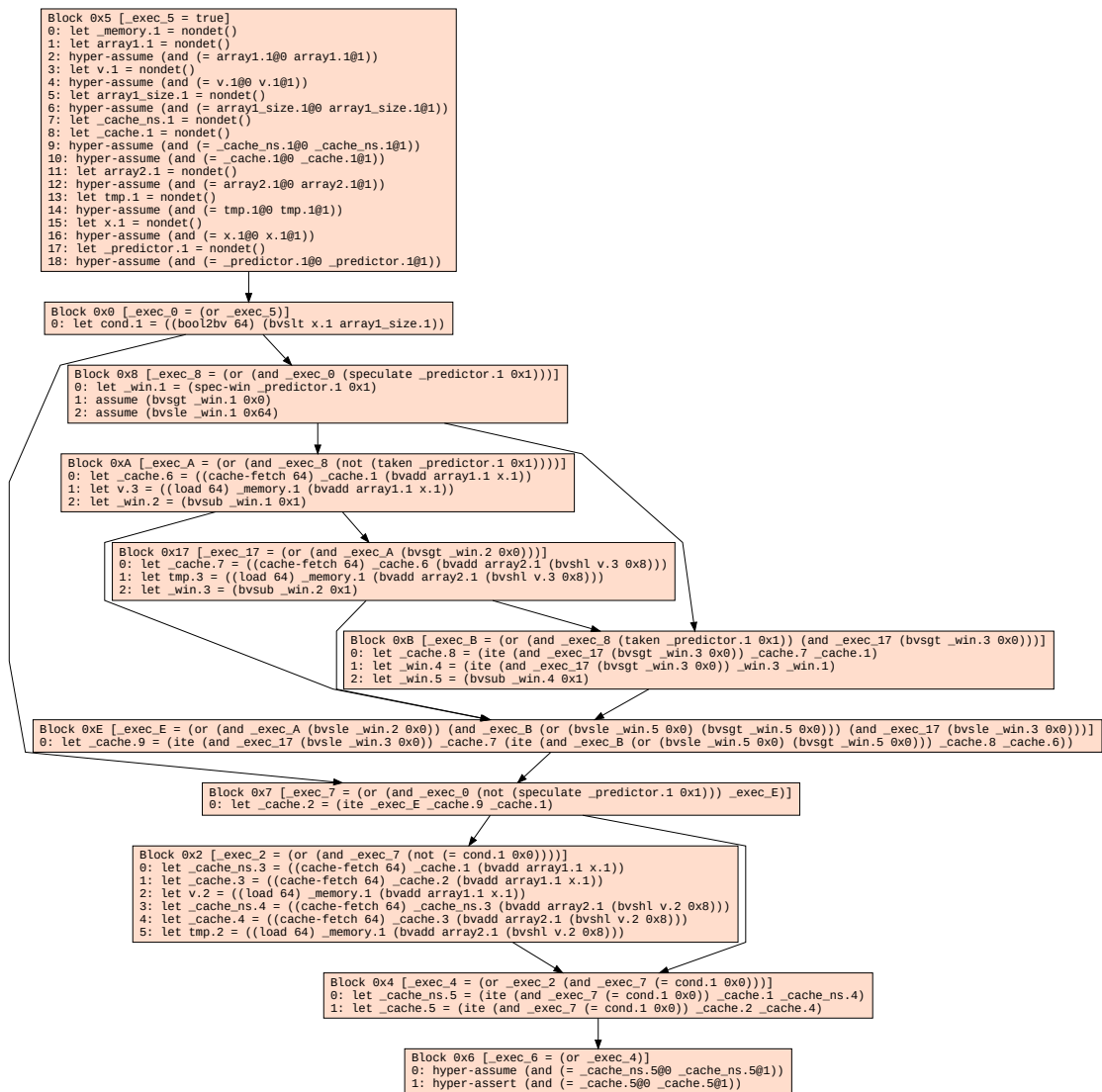


Figure 5.9: Mid-level Intermediate Representation of Kocher01 Example

bindings, assertions and assumptions, which makes the SMT encoding of LIR relatively straightforward. A LIR program is just a list of variable bindings, assertions and assumptions.

```

enum Node {
  Let { var: Variable, expr: Expression },
  Assert { cond: Expression },
  Assume { cond: Expression },
}

```

```

struct Program {
  nodes: Vec<Node>,
}

```

Listing 5.5: Low-level Intermediate Representation

The simplicity of LIR also allows us to easily perform a great variety of different optimizations respectively simplifications before handing the program over to the solver. Currently the following optimizations are implemented:

- **Constant Propagation:** Propagates all constant assignments but doesn't remove them. Constant assignments are defined as `let x = c` where `c` is a constant. For example, suppose that we have `let x = c` and `let y = x`, then constant propagation will give `let x = c` and `let y = c`.
- **Copy Propagation:** Propagates all simple assignments but doesn't remove them. Simple assignments are defined as `let x = v` where `v` is a variable. For example, suppose that we have `let x = v` and `let y = x`, then constant propagation will give `let x = v` and `let y = v`.
- **Expression Simplification:** Tries to simplify expressions, e.g., `x ^ false` will become `false`. In total, approximately 50 simplification rules have been implemented.
- **Constant Folding:** Tries to evaluate expressions to constants if all their operands are constant, e.g. `1 + 2` will become `3`. In total, approximately 20 folding rules have been implemented.
- **Dead Code Elimination:** Removes unused variable bindings. Implemented as a mark-and-sweep algorithm which in the first phase marks all assertions and assumptions including their dependencies, and in the second phase removes all unmarked variable bindings.
- **Assertion Elimination:** Removes assertions whose condition is already assumed. For example, suppose we have a program with `assume (x)` and `assert (x)`, then the assertion always holds and can therefore safely be removed.
- **Redundant Node Elimination:** Removes duplicated assumptions and assertions from the program.

Lowering (MIR \rightarrow LIR)

Lowering of MIR into LIR consists of two parts. One part is the flattening of the block graph to a list of nodes. The other part is to transform the 2-safety problem into a safety problem by means of self-composition [6]. The flattening of MIR into LIR is done as follows:

- The execution condition e_i of each block B_i is made explicit. This is achieved by introducing a fresh boolean variable `exec_ B_i` and bind the execution condition e_i to it, i.e., `LIR::Let {exec_ B_i , e_i }`.

- Variable bindings are lowered as is.

MIR::Let{var, expr} ▶ LIR::Let{var, expr}

- The condition `cond` should only be asserted if the assertion is actually executed, meaning that an assertion should be lowered s.t. it trivially holds if not executed. This is done by adding the execution condition `exec_Bi` of the surrounding block `Bi` as premise.

MIR::Assert{cond} ▶ LIR::Assert{exec_B_i ⇒ cond}

- Similar to assertions, the block's execution condition is added as premise.

MIR::Assume{cond} ▶ LIR::Assume{exec_B_i ⇒ cond}

After flattening, the list of nodes is duplicated by means of eager self-composition. During this process all variables are renamed by assign the composition ID to each copy of the variable (indicated by the @ID suffix). After self-composition the hyperproperties are lowered as follows:

- Let `Bi` denote the block surrounding the hyper-assertion. By self-composition we have two execution conditions `exec_Bi@0` and `exec_Bi@1`. Given that the condition `cond` should only be asserted if the hyper-assertion is actually executed in all compositions, we add `exec_Bi@0 ∧ exec_Bi@1` as premise.

MIR::HyperAssert{cond} ▶
LIR::Assert{exec_B_i@0 ∧ exec_B_i@1 ⇒ cond}

- Similar to hyper-assertions, the block's execution conditions are added as premise.

MIR::HyperAssume{cond} ▶
LIR::Assume{exec_B_i@0 ∧ exec_B_i@1 ⇒ cond}

Listing 5.6 shows an excerpt of the Kocher01 example lowered to LIR. Line 8 shows the explicit execution condition of block 0xA, the self-composition copy of it is shown at line 29. An example of a lowered hyper-assumption can be found at line 49 and an example for a lowered hyper-assertion is shown at line 50.

```

1 // Block 0x8@0
2 let _exec_8@0 = (speculate _predictor.1@0 0x1:64)
3 let _win.1@0 = (spec-win _predictor.1@0 0x1:64)
4 assume (=> _exec_8@0 (bvsgt _win.1@0 0x0:10))
5 assume (=> _exec_8@0 (bvsls _win.1@0 0x64:10))
6
7 // Block 0xA@0
8 let _exec_A@0 = (and _exec_8@0 (not (taken _predictor.1@0 0x1:64)))
9 let _cache.6@0 = ((cache-fetch 64) _cache.1@0 (bvadd array1.1@0 x.1@0))
10 let v.3@0 = ((load 64) _memory.1@0 (bvadd array1.1@0 x.1@0))
11 let _win.2@0 = (bvsub _win.1@0 0x1:10)
12
13 // Block 0xB@0
14 let _exec_B@0 =
15   (or (and _exec_8@0 (taken _predictor.1@0 0x1:64))
16     (and _exec_17@0 (bvsgt _win.3@0 0x0:10)))
17 let _cache.8@0 =
18   (ite (and _exec_17@0 (bvsgt _win.3@0 0x0:10)) _cache.7@0 _cache.1@0)

```



```

19 let _win.4@0 = (ite (and _exec_17@0 (bvsgt _win.3@0 0x0:10)) _win.3@0 _win.1@0)
20 let _win.5@0 = (bvsub _win.4@0 0x1:10)
21
22 // Block 0x8@1
23 let _exec_8@1 = (speculate _predictor.1@1 0x1:64)
24 let _win.1@1 = (spec-win _predictor.1@1 0x1:64)
25 assume (=> _exec_8@1 (bvsgt _win.1@1 0x0:10))
26 assume (=> _exec_8@1 (bvsls _win.1@1 0x64:10))
27
28 // Block 0xA@1
29 let _exec_A@1 = (and _exec_8@1 (not (taken _predictor.1@1 0x1:64)))
30 let _cache.6@1 = ((cache-fetch 64) _cache.1@1 (bvadd array1.1@1 x.1@1))
31 let v.3@1 = ((load 64) _memory.1@1 (bvadd array1.1@1 x.1@1))
32 let _win.2@1 = (bvsub _win.1@1 0x1:10)
33
34 // Block 0xB@1
35 let _exec_B@1 =
36   (or (and _exec_8@1 (taken _predictor.1@1 0x1:64))
37     (and _exec_17@1 (bvsgt _win.3@1 0x0:10)))
38 let _cache.8@1 =
39   (ite (and _exec_17@1 (bvsgt _win.3@1 0x0:10)) _cache.7@1 _cache.1@1)
40 let _win.4@1 = (ite (and _exec_17@1 (bvsgt _win.3@1 0x0:10)) _win.3@1 _win.1@1)
41 let _win.5@1 = (bvsub _win.4@1 0x1:10)
42
43 // Self-Composition Constraints
44 assume (= _predictor.1@0 _predictor.1@1)
45 assume (= _cache_ns.1@0 _cache_ns.1@1)
46 assume (= _cache.1@0 _cache.1@1)
47 assume (= array1.1@0 array1.1@1)
48 assume (= array2.1@0 array2.1@1)
49 assume (=> (and _exec_4@0 _exec_4@1) (= _cache_ns.5@0 _cache_ns.5@1))
50 assert (=> (and _exec_4@0 _exec_4@1) (= _cache.5@0 _cache.5@1))

```

Listing 5.6: Low-level Intermediate Representation of Kocher01 Example (Excerpt)

5.3.4 SMT Encoding

Encoding LIR as a satisfiability problem is straightforward. The current implementation makes use of the *rsmt2* library¹⁰, a generic library to interact with SMT-LIB 2 compliant solvers. Thereby, the solver runs as separate process and communicates via Unix pipes. SpecBMC uses Yices2 as the default solver, but Z3 and CVC4 are also available. The SMT encoding of LIR is done as follows:

- Variable bindings `LIR::Let{var, expr}` are encoded as constants. The type of the constant, indicated as *«type»*, is given by the type of the variable `var`.

```

(declare-const var <<type>>)
(assert (= var expr))

```

- Assumptions `LIR::Assume{cond}` are encoded as assertions.

```

(assert cond)

```

¹⁰<https://github.com/kino-mc/rsmt2>

- Encoding assertions `LIR::Assert{cond}` is a little bit more involved, as we want to find assertion violations. Meaning that the resulting formula should be satisfiable when an assertion is violated, but unsatisfiable if all assertions hold. This can be achieved by asserting the negation of the condition `cond`. As the program may contain multiple assertions, a single `assert` command with the negated conjunction of all assertions' condition is added.

For example, suppose that a program contains two assertions, `LIR::Assert{a}` and `LIR::Assert{b}`, then the encoding will yield the following:

```
(define-const _assertion0 Bool a)
(define-const _assertion1 Bool b)
(assert (not (and _assertion0 _assertion1)))
```

Listing 5.7 shows an excerpt of the Kocher01 example encoded as SMT formula. An example of an encoded variable binding can be found at line 13, an example of an encoded assumption can be found at line 15 and an example of an encoded assertion can be found at lines 42-44.

```
1 (declare-const _exec_8@0 Bool)
2 (declare-const _win.1@0 (_ BitVec 10))
3 (declare-const _exec_A@0 Bool)
4 (declare-const _cache.6@0 Cache)
5 (declare-const v.3@0 (_ BitVec 64))
6 (declare-const _win.2@0 (_ BitVec 10))
7 (declare-const _exec_B@0 Bool)
8 (declare-const _cache.8@0 Cache)
9 (declare-const _win.4@0 (_ BitVec 10))
10 (declare-const _win.5@0 (_ BitVec 10))
11
12 ; Block 0x8@0
13 (assert (= _exec_8@0 (pred-speculate _predictor.1@0 (_ bv1 64))))
14 (assert (= _win.1@0 (spec-win _predictor.1@0 (_ bv1 64))))
15 (assert (=> _exec_8@0 (bvsgt _spec_win.1@0 (_ bv0 10))))
16 (assert (=> _exec_8@0 (bvslc _spec_win.1@0 (_ bv100 10))))
17
18 ; Block 0xA@0
19 (assert (= _exec_A@0 (and _exec_8@0 (not (pred-taken _predictor.1@0 (_ bv1 64))))))
20 (assert (= _cache.6@0 (cache-fetch64 _cache.1@0 (bvadd array1.1@0 x.1@0))))
21 (assert (= v.3@0 (mem-load64 _memory.1@0 (bvadd array1.1@0 x.1@0))))
22 (assert (= _win.2@0 (bvsub _win.1@0 (_ bv1 10))))
23
24 ; Block 0xB@0
25 (assert (= _exec_B@0
26   (or (and _exec_8@0 (pred-taken _predictor.1@0 (_ bv1 64)))
27     (and _exec_17@0 (bvsgt _win.3@0 (_ bv0 10)))))
28 (assert (= _cache.8@0
29   (ite (and _exec_17@0 (bvsgt _win.3@0 (_ bv0 10))) _cache.7@0 _cache.1@0)))
30 (assert (= _win.4@0
31   (ite (and _exec_17@0 (bvsgt _win.3@0 (_ bv0 10))) _win.3@0 _win.1@0)))
32 (assert (= _win.5@0 (bvsub _win.4@0 (_ bv1 10))))
33
34 ; Self-Composition Constraints
35 (assert (= _predictor.1@0 _predictor.1@1))
36 (assert (= _cache_ns.1@0 _cache_ns.1@1))
37 (assert (= _cache.1@0 _cache.1@1))
38 (assert (= array1.1@0 array1.1@1))
```

```

39 (assert (= array2.1@0 array2.1@1))
40 (assert (=> (and _exec_4@0 _exec_4@1) (= _cache_ns.5@0 _cache_ns.5@1)))
41
42 (define-const _assertion0 Bool
43   (=> (and _exec_4@0 _exec_4@1) (= _cache.5@0 _cache.5@1)))
44 (assert (not (and _assertion0)))

```

Listing 5.7: SMT Formula of Kocher01 Example (Excerpt)

As told in the beginning of this section, SpecBMC provides an extensive expression library which abstractions for lists, tuples, memory and microarchitectual components. All these abstractions aren't part of SMT-LIB and need therefore be encoded in SMT as well. In the following the relevant SMT encodings are presented. The actual type of the abstract Word type depends on the architecture to be analyzed, e.g., for 64-bit architectures Word would be instantiated with (`_ BitVec 64`).

- **Predictor:** The predictor, as defined in Section 3.3, is encoded using the theory of uninterpreted functions.

```

1 (declare-sort Predictor 0)
2 (declare-fun spec-win (Predictor Word) (_ BitVec 10))
3 (declare-fun pred-speculate (Predictor Word) Bool)
4 (declare-fun pred-taken (Predictor Word) Bool)

```

- **Memory:** The byte-based memory [48] is encoded using the theory of arrays. Multiple functions for loading respectively storing in different bit widths are provided. They are simply mapped into a sequence of array stores and selects depending on the number of bytes to be stored or loaded. For example, the `mem-load16` function selects the bytes at index `addr` and `addr+1` from the memory array and concatenates them into a single 2-byte value. Storing works similar, e.g., `mem-store16` splits the 2-byte value into two individual bytes and stores the first byte at index `addr` and the second byte at index `addr+1`.

```

1 (define-sort Memory () (Array Word (_ BitVec 8)))
2
3 (define-fun mem-load8 ((mem Memory) (addr Word)) (_ BitVec 8)
4   (select mem addr)
5 )
6
7 (define-fun mem-load16
8   ((mem Memory) (addr Word)) (_ BitVec 16)
9   (concat
10    (select mem (bvadd addr (_ bv1 64)))
11    (select mem addr))
12 )
13
14 (define-fun mem-load32 ...)
15 (define-fun mem-load64 ...)
16 (define-fun mem-load128 ...)
17 (define-fun mem-load256 ...)

```

```

18 (define-fun mem-load512 ...)
19
20 (define-fun mem-store8
21   ((mem Memory) (addr Word) (v (_ BitVec 8))) Memory
22   (store mem addr v)
23 )
24
25 (define-fun mem-store16
26   ((mem Memory) (addr Word) (v (_ BitVec 16))) Memory
27   (store
28     (store mem (bvadd addr (_ bv1 64)) ((_ extract 15 8) val))
29     addr v)
30 )
31
32 (define-fun mem-store32 ...)
33 (define-fun mem-store64 ...)
34 (define-fun mem-store128 ...)
35 (define-fun mem-store256 ...)
36 (define-fun mem-store512 ...)

```

- **Cache:** The cache is implemented as defined in Section 4.1. The set semantics of the cache is given by the array of booleans, meaning that an address x is in the set if the array is `true` at index x . The only difference compared to the formalization is, that similar to the memory, the cache also addresses individual bytes. Therefore, multiple cache fetch functions for different bit widths are provided.

```

1 (define-sort Cache () (Array Word Bool))
2
3 (define-fun cache-fetch8 ((cache Cache) (addr Word)) Cache
4   (store cache addr true)
5 )
6
7 (define-fun cache-fetch16 ((cache Cache) (addr Word)) Cache
8   (store (store cache addr true) (bvadd addr (_ bv1 64)) true)
9 )
10
11 (define-fun cache-fetch32 ...)
12 (define-fun cache-fetch64 ...)
13 (define-fun cache-fetch128 ...)
14 (define-fun cache-fetch256 ...)
15 (define-fun cache-fetch512 ...)

```

- **Branch Target Buffer:** The BTB is implemented as defined in Section 4.3.4, meaning that for each program location the branch target is tracked in an array.

```

1 (define-sort BTB () (Array Word Word))
2
3 (define-fun btb-track ((btb BTB) (loc Word) (target Word)) BTB
4   (store btb loc target)
5 )

```

- **Pattern History Table:** The PHT is implemented as defined in Section 4.3.1, meaning that for each program location the branch decision is tracked in an array.

```

1 (define-sort PHT () (Array Word Bool))
2
3 (define-fun pht-taken ((pht PHT) (location Word)) PHT
4   (store pht location true)
5 )
6
7 (define-fun pht-not-taken ((pht PHT) (location Word)) PHT
8   (store pht location false)
9 )

```

- **List:** Lists are implemented using parametric recursive datatypes. The empty list is represented by `nil`. The `cons`¹¹ function, called the list constructor, is used to build a new list given an element `head` and an existing list `tail`. Note that Z3 has builtin support for the theory of lists, therefore this declaration is only necessary for CVC4.

```

1 (declare-datatypes ((List 1)) ((par (T) (
2   (nil)
3   (cons (head T) (tail (List T)))
4 )))

```

- **Tuple:** Tuples are a heterogeneous product of types. They are build using the tuple constructor `tuple<n>`, where n defines the number of fields of the tuple. Individual fields of a tuple can be accessed by the `tuple<n>-field<i>` function, where i , with $0 \leq i < n$, defines the index of the field to access. As shown in Listing 5.7 at line 4, tuples are used to hold the microarchitectual state in the trace.

```

1 (declare-datatypes ((Tuple1 1)) ((par (T0) (
2   (tuple1 (tuple1-field0 T0))
3 )))
4
5 (declare-datatypes ((Tuple2 2)) ((par (T0 T1) (
6   (tuple2 (tuple2-field0 T0) (tuple2-field1 T1))
7 )))
8
9 (declare-datatypes ((Tuple3 3)) ...)
10 ...
11 (declare-datatypes ((Tuple9 9)) ...)

```

5.4 Solver

After the encoding is done, the resulting SMT formula is handed over to an SMT solver. By emitting a single `check-sat` command, we request the solver to find a satisfiable

¹¹In Z3's builtin List type the `cons` function is called `insert` instead.

assignment for the encoded program. After some time, which can heavily vary depending on the structure and size of the program, the solver returns a result. If the result is *unsatisfiable*, meaning that all (LIR) assertions hold, the program is considered as **safe**¹². If the result is *satisfiable*, meaning that an (LIR) assertion has been violated, the program is considered as **insecure**. For insecure programs, the solver’s interpretation that made the formula satisfiable can be used to construct a counterexample.

5.4.1 Counterexample

A counterexample is represented as an annotated HIR program. The construction of a counterexample is roughly done as follows:

1. The execution paths of both compositions are reconstructed. This is done by starting at the entry block and walking along the edges whose conditions evaluate to `true`. We denote the execution path of the first composition as *A* and the execution path of the second composition as *B*.
2. Once both execution paths are reconstructed, all variable assignments along those two paths are requested from the solver and added to the annotated program.
3. Finally, the annotated program is rendered as a DOT graph. The execution path *A* is rendered as thick red path and *B* is rendered as thick blue path. If both execution paths overlay each other, then only the red path is shown.

The annotations are shown beneath each instruction. We use prefix (i) «*Composition*»\$ to show instructions with all read variables instantiated (ii) «*Composition*»@ to show the values of written variables (iii) «*Composition*»# to show the evaluated effects. For example, let `z = load((bvadd x y))` be an instruction which is part of execution path *A* and let *x* be `0x1`, *y* be `0x1` and the memory content at location `0x3` be `0x42`. Then the annotations of this instruction for composition *A* would be:

- A\$ `z = load((bvadd 0x1 0x2))`
- A@ `z = 0x42`
- A# `cache_fetch(0x3)`

Back to our example. For the Kocher01 program given in Listing 5.1 the solver returns *satisfiable*, meaning that this program is insecure. Figure 5.10 shows the corresponding counterexample. The transient execution starts at program location 1, where the conditional branch is predicted to be not taken. This causes an out-of-bounds read at program location 2, where a secret value is loaded into *v*. We have that register *v* is `0x21450002407F08` in execution *A*, but `0x4092132218040004` in execution *B*. The interesting part is the subsequent load instruction at program location 3. Because the previously loaded secret value of *v* is used in the address calculation, the secret value is encoded into the cache, thereby causing a data leak. The transient execution data leak is indicated by `A# cache_fetch(0xA756060A40870800, 64)` and `B# cache_fetch(0x1824282004080400, 64)`.

¹²Safe for the given unwinding bound and recursion limit.

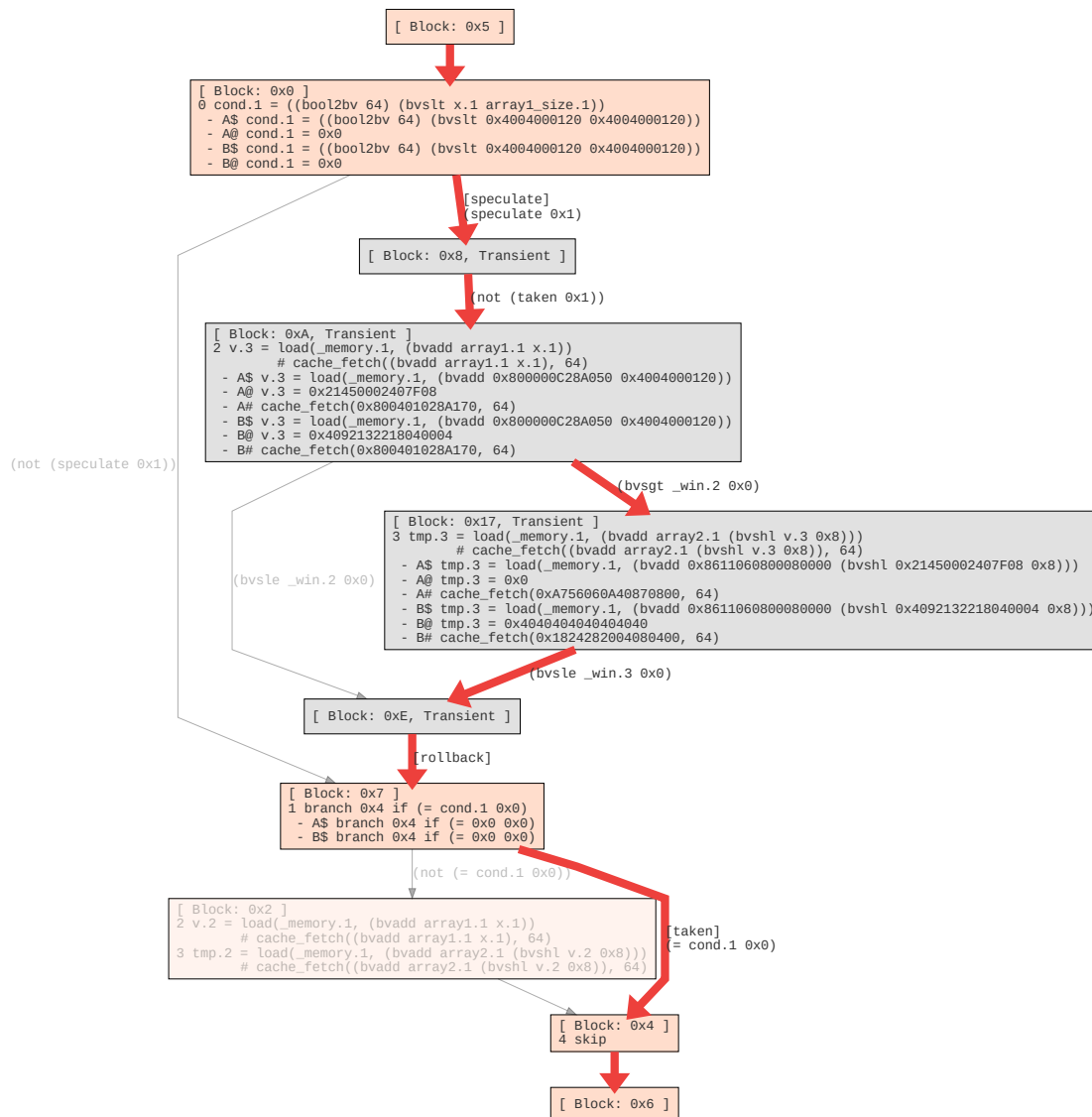


Figure 5.10: Counterexample for Koche01 Example

5.5 Additional Features

5.5.1 Program Counter Model

In addition to the components model as defined in Chapter 4, we implemented an enhanced version of the program counter security model [37] similar to Spectector’s observation model [21]. In the program counter (PC) model an adversary is able to observe the program counter and the accessed memory locations. In contrast to the components model, the side effects aren’t encoded into the microarchitectural state but instead

observed as is.

The model tracks the program counter state in a two elements tuple $\mathcal{PC} \subseteq \text{Word} \times \text{Word}$, where the first element is the program counter and the second element is the last accessed memory location. The derivation relation $\xrightarrow{\text{PC}}$ works on a configuration $\langle \mathcal{PC}, \varphi, \sigma, pc, \varphi', \sigma', pc' \rangle$, where \mathcal{PC} defines the current state of the PC model, φ the current register assignment, σ the current memory state, pc the current program counter, φ' the next register assignment, σ' the next memory state and pc' the next program counter. The evaluation is defined as follows, where load and store instructions update the observed memory location and branching instructions update the observed program counter.

$$\begin{aligned} \text{PCMEMORYACCESS} & \frac{\rho(pc) \in \{\text{load } x, e; \text{store } x, e\} \quad a'_{obs} = \llbracket e \rrbracket \varphi}{\langle (pc_{obs}, a_{obs}), \varphi, \sigma, pc, \varphi', \sigma', pc' \rangle \xrightarrow{\text{PC}} (pc_{obs}, a'_{obs})} \\ \text{PCBRANCH} & \frac{\rho(pc) \in \{\text{beqz } x, \ell; \text{jmp } \ell\} \quad pc'_{obs} = pc'}{\langle (pc_{obs}, a_{obs}), \varphi, \sigma, pc, \varphi', \sigma', pc' \rangle \xrightarrow{\text{PC}} (pc'_{obs}, a_{obs})} \\ \text{PCIGNORE} & \frac{\rho(pc) \notin \{\text{load } x, e; \text{store } x, e; \text{beqz } x, \ell; \text{jmp } \ell\}}{\langle \mathcal{PC}, \varphi, \sigma, pc, \varphi', \sigma', pc' \rangle \xrightarrow{\text{PC}} \mathcal{PC}} \end{aligned}$$

Figure 5.11 shows the implementation and application of the PC model to the Kocher01 example, using both the trace and parallel observations.

<pre> 1 trace = nil 2 c <- x < array1_size 3 4 pc = Ite(c=0, EndIf, Then) 5 trace = trace < (pc, _) 6 beqz c, EndIf 7 Then: 8 9 trace = trace < (pc, array1 + x) 10 load v, array1 + x 11 12 trace = trace < (pc, array2 + v) 13 load tmp, array2 + v 14 EndIf: 15 skip 16 observable(trace) </pre>	<pre> 1 2 c <- x < array1_size 3 4 pc = Ite(c=0, EndIf, Then) 5 observable(pc, _) 6 beqz c, EndIf 7 Then: 8 9 observable(pc, array1 + x) 10 load v, array1 + x 11 12 observable(pc, array2 + v) 13 load tmp, array2 + v 14 EndIf: 15 16 skip </pre>
---	---

(a) PC Model with Trace Observation

(b) PC Model with Parallel Observation

Figure 5.11: Program Counter Model applied to Kocher01 Example

Evaluation

In this chapter we evaluate SpecBMC and test it against different Spectre-STL/PHT examples and mitigations. Additionally, we conduct a small case study to run SpecBMC on a real-word program. In the following we let \checkmark indicate that the program is secure and \times that the program contains a leak. The hard- and software configuration of the test machine are given in Table 6.1. Compiler flags and SpecBMC specific settings are mentioned for each individual test.

Hardware	Intel Core i7-6700k, 40 GB DDR4-2133
Operating system	Arch Linux (Kernel 5.8.12)
Compiler	GCC (10.2.0), Clang (10.0.1)
Z3	4.8.7
Yices2	2.6.2 (Rev: 60aaae9e78a89dd56125f6aacfd46203b7ebf30e)
SpecBMC	Rev: 56749f8fe4f1207064ccebfa874159948217cd0
Spectector	Rev: 839bec7d96cd5b3387377cb42d26eb8ad55ecd4c

Table 6.1: Hard- and Software Configuration

6.1 Spectre-STL

We check if SpecBMC is able to detect Spectre-STL vulnerabilities. The example code shown in Listing 6.1 is taken from the Transient Fail project¹, which is a collection of proof-of-concepts for transient execution attacks developed by Canella et al. [7]. As explained in Section 3.1.3, Spectre-STL relies on mis-predicted data dependencies to bypass store instructions. The example in Listing 6.1 uses two pointer indirections to speculatively bypass the overwrite of the secret value. During transient execution, the

¹Transient Fail Project: <https://github.com/IAIK/transientfail>

secret value is then encoded into the cache using a standard cache encoding gadget as shown in Listing 6.2.

```

1  #define OVERWRITE '#'
2
3  char* data;
4
5  void access_array(int x) {
6      // store secret in data
7      strcpy(data, SECRET);
8
9      // flushing the data which is used in the condition increases
10     // probability of speculation
11     mfence();
12     char** data_slowptr = &data;
13     char*** data_slowslowptr = &data_slowptr;
14     mfence();
15     flush(&x);
16     flush(data_slowptr);
17     flush(&data_slowptr);
18     flush(data_slowslowptr);
19     flush(&data_slowslowptr);
20     // ensure data is flushed at this point
21     mfence();
22
23     // overwrite data via different pointer
24     // pointer chasing makes this extremely slow
25     (*(data_slowslowptr))[x] = OVERWRITE;
26 #ifdef FENCE_MITIGATION
27     mfence();
28 #endif
29     // data[x] should now be "#"
30
31     // Encode stale value in the cache
32     cache_encode(data[x]);
33 }

```

Listing 6.1: Spectre-STL Example

```

1 sym.cache_encode:
2 0x00001750 movsx rcx, dil
3 0x00001754 imul rcx, qword [obj.pagesize]
4 0x0000175c add rcx, qword [obj.mem]
5 0x00001763 mov rax, qword [rcx]
6 0x00001766 ret

```

Listing 6.2: Assembly Code of cache_encode Function

```

1 analysis:
2     spectre_pht: false

```

```

3   spectre_stl: true
4   check: only_transient_leaks
5   program_entry: "access_array"
6 architecture:
7   cache: true
8   btb: false
9   pht: false
10 policy:
11   registers:
12     default: low
13   memory:
14     default: high
15     low:
16       - # obj.data
17         start: 0x40e0
18         end: 0x40e9
19       - # obj.mem
20         start: 0x41c8
21         end: 0x41d1
22       - # obj.pagesize
23         start: 0x4258
24         end: 0x4261
25 setup:
26   init_stack: true

```

Listing 6.3: SpecBMC Environment for Spectre-STL Test

The example from Listing 6.1 is compiled with `gcc -o stl -O2 -I. main.c`, meaning that the fence at line 27 is initially disabled. SpecBMC is invoked with the environment defined in Listing 6.3. It's assumed that all register are low security and all memory content is high security by default. Additionally, it's assumed that the data as well as the mem pointer and the value of pagesize is known to the attacker. In this test the check is limited to cache-based covert channels only.

SpecBMC detects a leak at program location `0x1763`, which is the memory load instruction in the cache encode gadget shown in Listing 6.2. As the address depends on the value loaded from `data[x]`, we have that the bypassed store leaks secret information when loaded at line 32. SpecBMC provides the following counterexample for the encoded secret value at program location `0x1763`:

```

1 0x1763 rax = load(_memory, rcx)
2   - A# cache_fetch(0xFFFFFFFFFCBC010, 64)
3   - B# cache_fetch(0x9FFFE2000000002F, 64)

```

6.1.1 Mitigation

As explained in Section 3.1.3, fences can be used to mitigate Spectre-STL vulnerabilities. After enabling the fence mitigation at line 27 of Listing 6.1, SpecBMC marks the function as secure. The example from Listing 6.1 is compiled with `gcc -o stl_fence`

-O2 -I. -DFENCE_MITIGATION main.c and run with the same environment as the unmitigated example.

6.1.2 Summary

The evaluation results for Spectre-STL are summarized in Table 6.2. SpecBMC is able to detect Spectre-STL vulnerabilities and shows that fences are successful in mitigation them.

Mitigation	
None	✗
Fence	✓

Table 6.2: Evaluation Results for Spectre-STL + Mitigation

6.2 Spectre-PHT

We check if SpecBMC is able to detect Spectre-PHT vulnerabilities. The example code shown in Listing 6.4 is adopted from the proof-of-concept of Kocher et al. [28] and SpectrePoC². As described in Section 3.1.2, the index masking approach may not fully mitigate the vulnerability. To test if SpecBMC is able to detect potential index masking problems, we additionally run the test on the example shown in Listing 6.5. The only difference compared to Listing 6.4 is, that the array size isn't a power of two.

```

1 size_t array1_size = 16;
2 uint8_t array1[16] = {1, 2, 3, 4, ..., 15, 16};
3
4 void victim_function(size_t x) {
5     if (x < array1_size) {
6         y &= array2[array1[x] * 512];
7     }
8 }
```

Listing 6.4: Spectre-PHT Example (Array of Size 16)

```

1 size_t array1_size = 15;
2 uint8_t array1[15] = {1, 2, 3, 4, ..., 14, 15};
3
4 void victim_function(size_t x) {
5     if (x < array1_size) {
6         y &= array2[array1[x] * 512];
7     }
8 }
```

Listing 6.5: Spectre-PHT Example (Array of Size 15)

²SpectrePoC: <https://github.com/crozone/SpectrePoC>

```

1 analysis:
2   spectre_pht: true
3   spectre_stl: false
4   check: only_transient_leaks
5   program_entry: "victim_function"
6 architecture:
7   cache: true
8   btb: false
9   pht: false
10 policy:
11   registers:
12     default: low
13   memory:
14     default: high
15     low:
16       - # array1
17         start: 0x4050
18         end: 0x4060    # 0x405f for array size 15
19       - # array1_size
20         start: 0x4060
21         end: 0x4065

```

Listing 6.6: SpecBMC Environment for Spectre-PHT Test

The examples shown in Listings 6.4 and 6.5 are compiled with `gcc -o pht -O2 main.c`. The assembly output of the victim function is depicted in Figure 6.1. SpecBMC is invoked with the environment defined in Listing 6.6. It's assumed that all registers are low security and all memory content is high security by default. Additionally, it's assumed that the `array1` pointer and the value of `array1_size` is known to the attacker. In this test the check is limited to cache-based covert channels only.

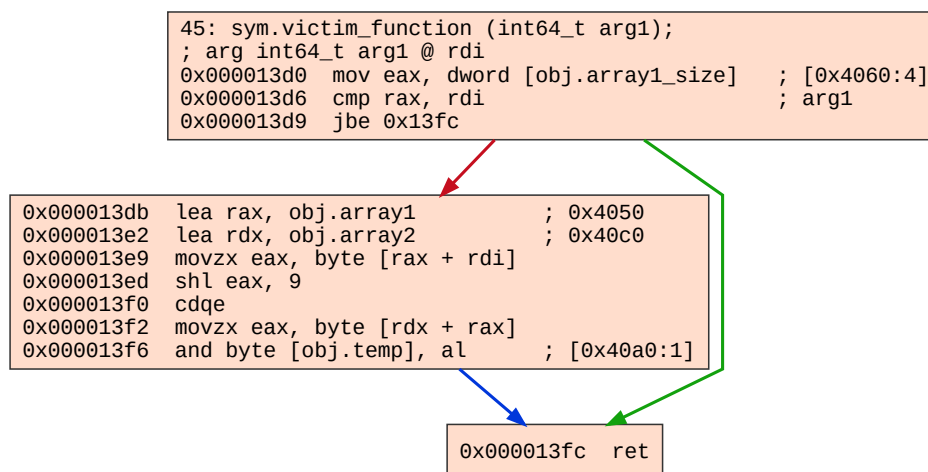


Figure 6.1: Assembly Code of Spectre-PHT Victim Function

SpecBMC detects a leak at program location `0x13F2`, which is the memory load from `array2`. As the address depends on the value loaded from `array1[x]`, we have that the memory load leaks secret information when the bounds check is speculatively bypassed. SpecBMC provides the following counterexample for the program shown in Listing 6.4, where the index variable `x`, located in register `rdi`, has value `0x2000000000000089`. The encoding of the secret value into the cache can be seen at lines 13 and 14.

```

1 0x13E9 rax = ((bvzext 56) load(_memory, (bvadd 0x4050 rdi)))
2   - A# cache_fetch(0x20000000000040D9, 8)
3   - B# cache_fetch(0x20000000000040D9, 8)
4   - A@ rax = 0x10
5   - B@ rax = 0x0
6 0x13ED rax = ((bvzext 32) (bvshl ((bvtrunc 32) rax) 0x9))
7   - A@ rax = 0x2000
8   - B@ rax = 0x0
9 0x13F0 rax = ((bvsext 32) ((bvtrunc 32) rax))
10  - A@ rax = 0x2000
11  - B@ rax = 0x0
12 0x13F2 rax = ((bvzext 56) load(_memory, (bvadd 0x40C0 rax)))
13  - A# cache_fetch(0x60C0, 8)
14  - B# cache_fetch(0x40C0, 8)

```

6.2.1 Mitigation

As described in Section 3.1.2, multiple proposed mitigations for Spectre-PHT vulnerabilities exist. We test each of them on the examples shown in Listings 6.4 and 6.5.

Speculative Load Hardening

When compiling the examples from Listings 6.4 and 6.5 with LLVM's speculative load hardening, the victim function is marked as secure, both for array size 15 and 16. The compiler is invoked with `clang -o pht_slh -mspeculative-load-hardening -O2 main.c`.

Fence

Inserting a memory fence prevents the CPU from speculating [25] and therefore should make the load unreachable when speculatively bypassing the bounds check. Indeed, after inserting the fence before the load as shown in Listing 6.7, SpecBMC marks the victim function as secure, both for array size 15 and 16.

```

1 void victim_function(size_t x) {
2   if (x < array1_size) {
3     mfence();
4     y &= array2[array1[x] * 512];
5   }
6 }

```

Listing 6.7: Spectre-PHT Example Mitigated by Fence

Linux Kernel Mitigation

Linux uses a mitigation approach similar to SLH, but instead of relying on the compiler to insert the required mitigations, the Linux kernel mitigations are manually inserted by the developers. Linux's mitigation gadget³ is shown in Listing 6.8. Instead of using conditional moves as in SLH, the subtract instruction `sbb` is used to obtain the index mask. The developers are supposed to protect all vulnerable code sequences with the `array_index_nospec`⁴ macro. Note that this mitigation approach has also been used in our motivating example from Section 1.1.1.

```

1 /**
2  * generate a mask that is ~0UL when the bounds check succeeds
3  * and 0 otherwise. Returns: 0 - (index < size)
4  */
5 static inline unsigned long
6 array_index_mask_nospec(unsigned long index, unsigned long size) {
7     unsigned long mask;
8     asm volatile ("cmp_%1,%2;_sbb_%0,%0;"
9         : "=r" (mask)
10        : "g" (size), "r" (index)
11        : "cc");
12     return mask;
13 }
14
15 /**
16  * sanitize an array index after a bounds check
17  */
18 #define array_index_nospec(index, size) ({
19     unsigned long _mask = array_index_mask_nospec(index, size); \
20     (index & _mask); \
21 })

```

Listing 6.8: Linux Kernel Spectre-PHT Mitigation

When adding the Linux kernel mitigation to the victim function as shown in Listing 6.9, the victim function is marked as secure, both for array size 15 and 16.

```

1 void victim_function(size_t x) {
2     if (x < array1_size) {
3         x = array_index_nospec(x, array1_size);
4         y &= array2[array1[x] * 512];
5     }
6 }

```

Listing 6.9: Spectre-PHT Example Mitigated by Linux Kernel Mitigation

³<https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/x86/include/asm/barrier.h>

⁴<https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/linux/nospec.h>

Index Masking

As described in Section 3.1.2, index masking applies bit masks to index variables to protect from arbitrary out-of-bounds reads. As the examples have an array of size 15 respectively 16, the necessary mask is `0x0F`. When adding the index mask to the victim function as shown in Listing 6.10, the victim function is marked as secure, but only for array size 16. For array size 15, SpecBMC finds a counterexample where the index variable `x` has value 15, causing an speculative out-of-bounds byte read at `array1[15]`.

```

1 void victim_function(size_t x) {
2   if (x < array1_size) {
3     x &= 0x0F;
4     y &= array2[array1[x] * 512];
5   }
6 }

```

Listing 6.10: Spectre-PHT Example Mitigated by Index Masking

6.2.2 Summary

The evaluation results for Spectre-PHT are summarized in Table 6.3. SpecBMC is able to detect Spectre-PHT vulnerabilities and shows that fences, SLH and the Linux Kernel mitigation are successful in mitigation them. As expected, the index masking mitigation still allows to leak sensitive information if the mask is too lax, albeit only within the range of the mask.

Array Size	15	16
Mitigation		
None	✗	✗
Fence	✓	✓
SLH	✓	✓
Linux Kernel Mitigation	✓	✓
Index Masking	✗	✓

Table 6.3: Evaluation Results for Spectre-PHT + Mitigation

6.3 Kocher Examples

We check if SpecBMC is able to detect Spectre-PHT vulnerabilities in all the 15 Kocher examples, using different compiler optimization levels and mitigations. The C sources of all 15 examples have been taken from the Spectector benchmark repository⁵. We use the LLVM compiler for all examples. Each example is compiled with `O0` and `O2` optimization level. The fence and SLH mitigations are automatically inserted by the compiler. For fence mitigation we use the additional `-mspeculative-load-hardening`

⁵<https://github.com/spectector/spectector-benchmarks>

-mllvm -x86-slh-lfence compiler flags. For SLH mitigation we use the additional -mspeculative-load-hardening -mllvm -x86-slh-indirect compiler flags.

```

1 analysis:
2   spectre_pht: true
3   spectre_stl: false
4   check: only_transient_leaks
5   unwind: 1
6   recursion_limit: 1
7 architecture:
8   cache: true
9   btb: false
10  pht: true
11 policy:
12  registers:
13    default: low
14  memory:
15    default: high
16    low:
17      - # array1_size
18        start: 0x4030
19        end: 0x4035
20 setup:
21  init_stack: true

```

Listing 6.11: SpecBMC Environment for Kocher Example 1-14

```

1 policy:
2   registers:
3     default: low
4   memory:
5     default: high
6     low:
7       - # array1_size
8         start: 0x4030
9         end: 0x4035
10      - # *x (see rdi)
11        start: 0xffff0000
12        end: 0xffff0009
13 setup:
14  init_stack: true
15  registers:
16  rdi: 0xffff0000 # x pointer is first argument -> rdi

```

Listing 6.12: SpecBMC Environment for Kocher Example 15 (Reduced)

SpecBMC is invoked with the environment defined in Listing 6.11, expect for example 15 which uses the environment given in Listing 6.12. It's assumed that all registers are low security and all memory content is high security by default. Additionally, it's assumed that the value of `array1_size` is known to the attacker. The difference of the

environment shown in Listing 6.12 compared to Listing 6.11 is, that we additionally set the initial value of register `rdi`, which holds the `x` pointer argument, to `0xffff0000` and assume that that word value at memory location `0xffff0000` is of low security. This is motivated by the fact, that the adversary is in control of the index, which is in example 15 passed on by pointer instead of value, as shown in Listing 6.15.

For the evaluation the execution time of SpecBMC is limited to 1800 seconds. In the following we use TO to denote a timeout, meaning that SpecBMC didn't terminate with the specified time limit. We run SpecBMC with trace, sequential and parallel observe using the components model. Additionally, we run SpecBMC using the program counter model with parallel observe, as this setting is most similar to how Spectector models the observations. For all checks we use Yices2 as solver, except for the trace observe setting where we use Z3.

The evaluation results for the different optimization levels and mitigations are summarized in Table 6.4. As expected for these examples, an adversary who can only observe sequentially, i.e., only after the victim function has finished, observes the same leaks as an adversary running in parallel to the victim. The results also indicate that the trace observe setting is much more time-consuming for the solver than the other settings. The reason for this may be the additional overhead of the theory of lists and tuples required to hold the intermediate states of the microarchitectural components. For example, when we only consider the cache, we have that the type of the trace is an array within a tuple within a list. Also interesting is, that only the program counter model is able to check all examples within the given time limit. This indicates that the encoding into the different microarchitectural components may be too expensive and the abstraction level should maybe be lifted to a higher-level, such as only considering the side-effects directly without encoding them into components' states first. Apart from the timeouts, all settings give the same secure/leak results, which match with Spectector's results [21].

6.3.1 Evaluation Results Analysis

We give a short explanation for each outlier seen in Table 6.4.

- **Example 8 (O2, None):** The reason for example 8 to be secure when compiled without any mitigation is, that the example has speculative load hardening quasi built-in. As shown in Listing 6.13, the `x < array1_size ? (x + 1) : 0` causes that the index is 0 when `x` is out of bounds. When the optimizations are disabled, the conditional expression is translated into branching instructions, thereby allowing the adversary to speculative bypass the protection. Whereas, the conditional expression is translated into branchless code when the optimizations are turned on. Therefore, the program leaks without optimization whereas the branchless code generated by the optimizer is secure.

```

1 void victim_function_v08(size_t x) {
2     temp &= array2[array1[x < array1_size ? (x + 1) : 0] * 512];
3 }

```

Listing 6.13: Kocher Example 08

Ex.	O0			O2		
	None	Fence	SLH	None	Fence	SLH
01	✗	✓	✓	✗	✓	✓
02	TO	✓	✓	✗	✓	✓
03	TO	✓	✓	✗	✓	✓
04	✗	✓	✓	✗	✓	✓
05	TO	✓	✓	TO	✓	✓
06	✗	✓	✓	✗	✓	✓
07	TO	✓	✓	✗	✓	✓
08	TO	✓	TO	✓	✓	✓
09	✗	✓	✓	✗	✓	✓
10	TO	✓	✓	✗	✓	TO
11	TO	✓	✓	✗	✓	✓
12	✗	✓	✓	✗	✓	✓
13	TO	✓	TO	✗	✓	✓
14	✗	✓	✓	✗	✓	✓
15	TO	✓	TO	✗	✓	✓

(a) Components Model, Observe Trace

Ex.	O0			O2		
	None	Fence	SLH	None	Fence	SLH
01	✗	✓	✓	✗	✓	✓
02	✗	✓	✓	✗	✓	✓
03	✗	✓	✓	✗	✓	✓
04	✗	✓	✓	✗	✓	✓
05	✗	✓	✓	✗	✓	✓
06	✗	✓	✓	✗	✓	✓
07	✗	✓	✓	✗	✓	✓
08	✗	✓	✓	✓	✓	✓
09	✗	✓	✓	✗	✓	✓
10	✗	✓	✓	✗	✓	✗
11	TO	✓	TO	✗	✓	✓
12	✗	✓	✓	✗	✓	✓
13	✗	✓	✓	✗	✓	✓
14	✗	✓	✓	✗	✓	✓
15	✗	✓	✗	✗	✓	✓

(b) Components Model, Observe Sequential

Ex.	O0			O2		
	None	Fence	SLH	None	Fence	SLH
01	✗	✓	✓	✗	✓	✓
02	✗	✓	✓	✗	✓	✓
03	✗	✓	✓	✗	✓	✓
04	✗	✓	✓	✗	✓	✓
05	✗	✓	✓	✗	✓	✓
06	✗	✓	✓	✗	✓	✓
07	✗	✓	✓	✗	✓	✓
08	✗	✓	✓	✓	✓	✓
09	✗	✓	✓	✗	✓	✓
10	✗	✓	✓	✗	✓	✗
11	✗	✓	TO	✗	✓	✓
12	✗	✓	✓	✗	✓	✓
13	✗	✓	✓	✗	✓	✓
14	✗	✓	✓	✗	✓	✓
15	✗	✓	✗	✗	✓	✓

(c) Components Model, Observe Parallel

Ex.	O0			O2		
	None	Fence	SLH	None	Fence	SLH
01	✗	✓	✓	✗	✓	✓
02	✗	✓	✓	✗	✓	✓
03	✗	✓	✓	✗	✓	✓
04	✗	✓	✓	✗	✓	✓
05	✗	✓	✓	✗	✓	✓
06	✗	✓	✓	✗	✓	✓
07	✗	✓	✓	✗	✓	✓
08	✗	✓	✓	✓	✓	✓
09	✗	✓	✓	✗	✓	✓
10	✗	✓	✓	✗	✓	✗
11	✗	✓	✓	✗	✓	✓
12	✗	✓	✓	✗	✓	✓
13	✗	✓	✓	✗	✓	✓
14	✗	✓	✓	✗	✓	✓
15	✗	✓	✗	✗	✓	✓

(d) Program Counter Model, Observe Parallel

Table 6.4: Evaluation Results of Kocher Examples for Different Settings

- **Example 10 (O2, SLH):** The reason for example 10 to be insecure with SLH mitigation when compiled with optimizations turned on is, that the nested conditional statement at line 3 in Listing 6.14 depends on the value of `array1[x]`, which can be a secret value in case of speculative bounds-check bypass. While the compiler properly masks the loaded value when compiled with O0, the compiler fails to mask the loaded value when compiled with O2. A PHT-based covert channel allows to reveal if the loaded secret value is equal to `k` or not.

```

1 void victim_function_v10(size_t x, uint8_t k) {
2     if (x < array1_size) {
3         if (array1[x] == k)
4             temp &= array2[0];
5     }
6 }

```

Listing 6.14: Kocher Example 10

SpecBMC provides the following counterexample for the O2 optimized assembly output shown in Figure 6.2. The index variable `x`, located in register `rdi`, has value `0x8000000000000000` and the variable `k`, located in register `rsi`, has value `0x0`. Transient execution begins at `0x1135` where the jump is predicted to be not taken. As seen in lines 4 and 7, the masking is properly applied to `array1` and index variable `x`. This prevents attacker-controlled loads from arbitrary memory locations, instead the only address which can be loaded during transient execution is `0xFFFFFFFFFFFFFFFE`. As the memory content is assumed to be high-security by default, the byte loaded from this address is a secret value. Because the compiler doesn't add masking for the loaded value, some information about the secret value can be leaked via the subsequent conditional branch instruction. The differing effects of the secret value onto the PHT can be seen at lines 19 and 20.

```

1 0x1137 rcx = (ite (or CF ZF) 0xFFFFFFFFFFFFFFF 0x0)
2   - A@ rcx = 0xFFFFFFFFFFFFFFF
3   - B@ rcx = 0xFFFFFFFFFFFFFFF
4 0x1142 rdi = (bvor rdi rcx)
5   - A@ rdi = 0xFFFFFFFFFFFFFFF
6   - B@ rdi = 0xFFFFFFFFFFFFFFF
7 0x1145 rdx = (bvor 0x4040 rcx)
8   - A@ rdx = 0xFFFFFFFFFFFFFFF
9   - B@ rdx = 0xFFFFFFFFFFFFFFF
10 0x1148 temp_0x1148 = load(_memory, (bvadd rdi rdx))
11   - A# cache_fetch(0xFFFFFFFFFFFFFFFE, 8)
12   - B# cache_fetch(0xFFFFFFFFFFFFFFFE, 8)
13   - A@ temp_0x1148 = 0x80
14   - B@ temp_0x1148 = 0x0
15 0x1148 ZF = (= (bvsb temp_0x1148 ((bvtrunc 8) rsi)) 0x0)
16   - A@ ZF = false
17   - B@ ZF = true
18 0x114C branch 0x1164 if (not ZF)

```

```

19 - A# branch_condition(0x114C, true)
20 - B# branch_condition(0x114C, false)

```

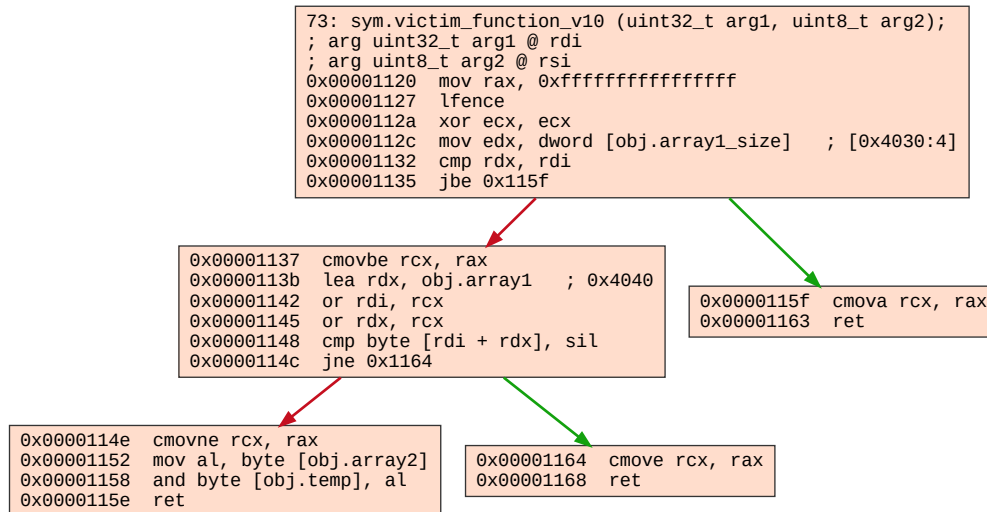


Figure 6.2: Assembly Code of Kocher Example 10 (O2, SLH)

- **Example 15 (O0, SLH):** The reason for example 15 to be insecure with SLH mitigation when compiled without optimizations is, that the compiler doesn't mask the loaded value of `array1[*x]` before using it as an index for the second load. The assembly code given in Figure 6.3 shows the problem. At line `0x118f` the value is loaded into register `edi`, which is then moved to `rax` at line `0x1196`. The register `rax` is then used in the second load at line `0x11a0`. Similar to example 10, the masking is properly applied to `array1` and index variable `*x`. This prevents attacker-controlled loads from arbitrary memory locations, instead the only address which can be loaded during transient execution is `0xFFFFFFFFFFFFFFFF`. But as the memory content is assumed to be high-security by default, the byte loaded from this address into register `edi` is a secret value which can be leaked through a cache-based covert channel. When compiled with optimizations turned on, this leak is eliminated.

```

1 void victim_function_v15(size_t *x) {
2     if (*x < array1_size)
3         temp &= array2[array1[*x] * 512];
4 }

```

Listing 6.15: Kocher Example 15

SpecBMC provides the following counterexample for the assembly output shown in Figure 6.3. As seen in lines 2 and 3, the memory load is limited to address `0xFFFFFFFFFFFFFFFF`. As the memory content is assumed to be high-security

by default, the byte loaded from this address is a secret value. The encoding of the secret value into the cache can be seen at lines 16 and 17.

```

1 0x118F temp_0x118F = load(_memory, (bvadd rax rsi))
2   - A# cache_fetch(0xFFFFFFFFFFFFFFFE, 8)
3   - B# cache_fetch(0xFFFFFFFFFFFFFFFE, 8)
4   - A@ temp_0x118F = 0xE0
5   - B@ temp_0x118F = 0xFF
6 0x118F rdi = ((bvzext 56) temp_0x118F)
7   - A@ rdi = 0xE0
8   - B@ rdi = 0xFF
9 0x1193 rdi = ((bvzext 32) (bvshl ((bvtrunc 32) rdi) 0x9))
10  - A@ rdi = 0x1C000
11  - B@ rdi = 0x1FE00
12 0x1196 rax = ((bvsext 32) ((bvtrunc 32) rdi))
13  - A@ rax = 0x1C000
14  - B@ rax = 0x1FE00
15 0x11A0 temp_0x11A0 = load(_memory, (bvadd 0x4060 rax))
16  - A# cache_fetch(0x20060, 8)
17  - B# cache_fetch(0x23E60, 8)

```

6.3.2 Runtime Analysis

All benchmarks are repeated at least 5 times and the execution time is measured using `hyperfine`⁶. The execution time of each run is limited to 300 seconds. The following graphs show the average execution time in seconds as well as the 95 % confidence interval.

In the first analysis we compare the performance of the program counter model with the components model, using parallel and trace observation. As the trace observation requires the Z3 solver, we run all the 15 benchmarks with Z3 only. The benchmark results are shown in Figures 6.4 to 6.6. The benchmark results affirm our assumption, that the trace observation is expensive in terms of execution time. Interestingly, the trace observation setting has a much lower execution time for example 11 (SLH, O0). Another clearly identifiable pattern is, that the examples using O2 optimization are usually much faster to check than their counterparts without optimizations. The reason may be, that the optimizations removes many stack operations and therefore the number of memory load and stores is much lower when optimization is turned on. Less memory load and stores means less array operations, which may help the SMT solver.

In the second analysis we compare the performance of SpecBMC to Spectector. We use the program counter model with parallel observe, as this setting is most similar to Spectector's observation model. This benchmark gives us a rough comparison between our BMC approach and Spectector's symbolic execution approach. We run the SpecBMC benchmarks using the Yices2 and Z3 solver, whereas Spectector uses Z3 only. The benchmark results are shown in Figures 6.7 to 6.9. The comparison shows that for O2 compiled binaries, the performance of SpecBMC and Spectector is pretty similar. For

⁶Hyperfine: <https://github.com/sharkdp/hyperfine>

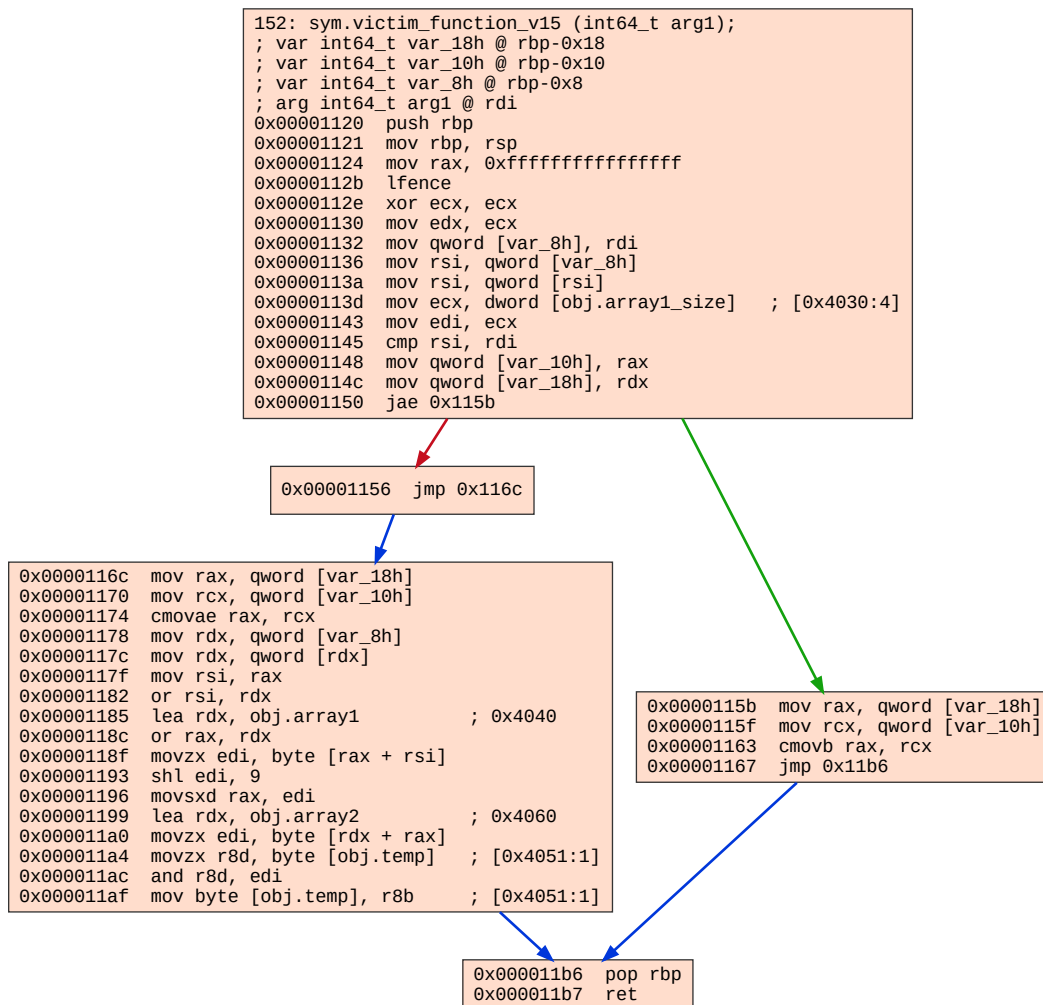


Figure 6.3: Assembly Code of Kocher Example 15 (O0, SLH)

O0 compiled binaries Spectector is in many cases much faster than SpecBMC with Z3. The reason may be that the byte-based memory implementation in SpecBMC is more expensive compared to Spectector’s memory model. If we compare only the SpecBMC results of Yices2 and Z3, we can see that Yices2 in many cases much faster than Z3. It comes as no surprise that Yices2 is faster, because Yices2 is the this year’s winner in the *QF_AUFBV* single query track SMT competition 2020⁷.

⁷<https://smt-comp.github.io/2020/results/qf-aufbv-single-query>

6. EVALUATION

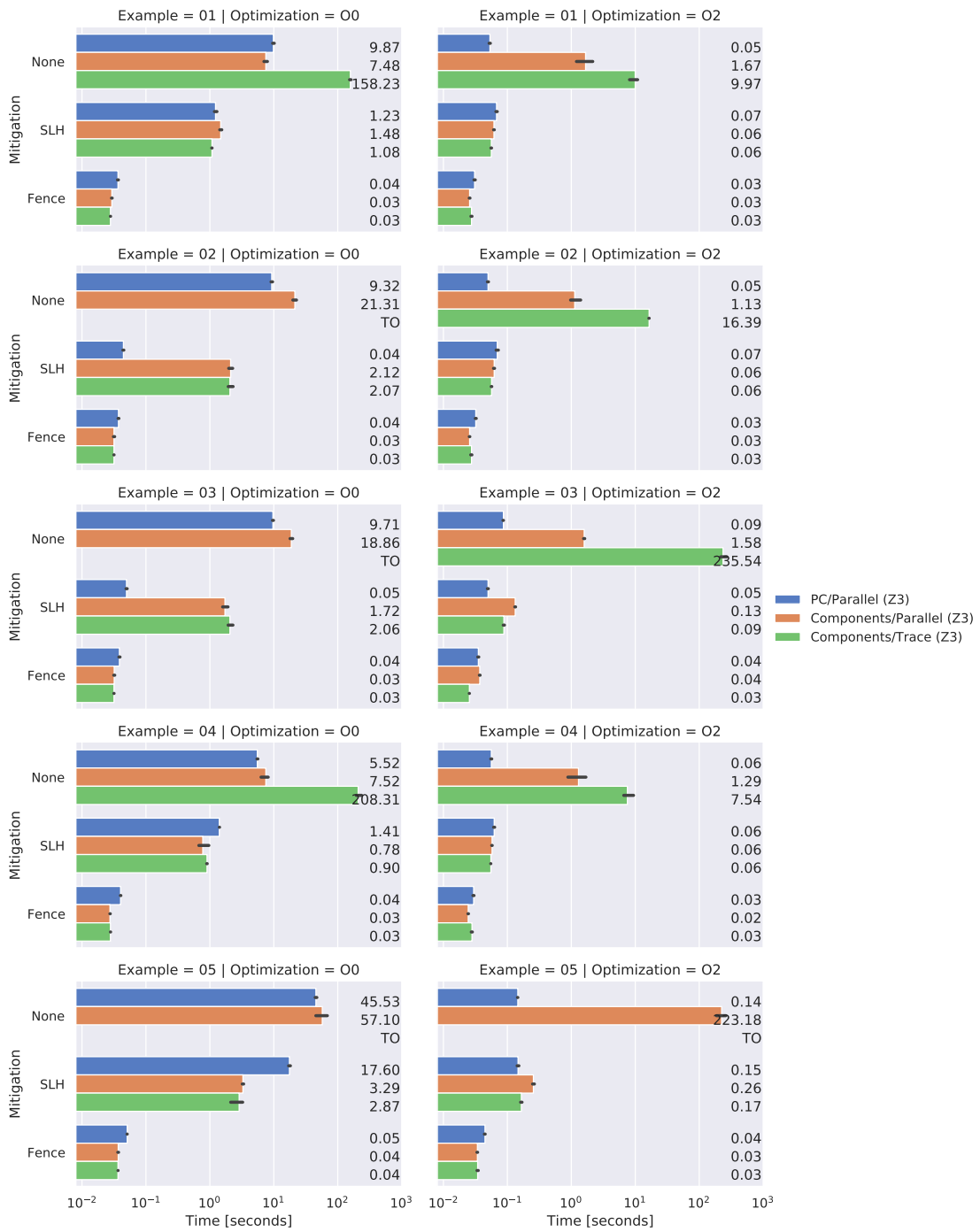


Figure 6.4: Execution Time of SpecBMC using different Settings (Examples 1-5)

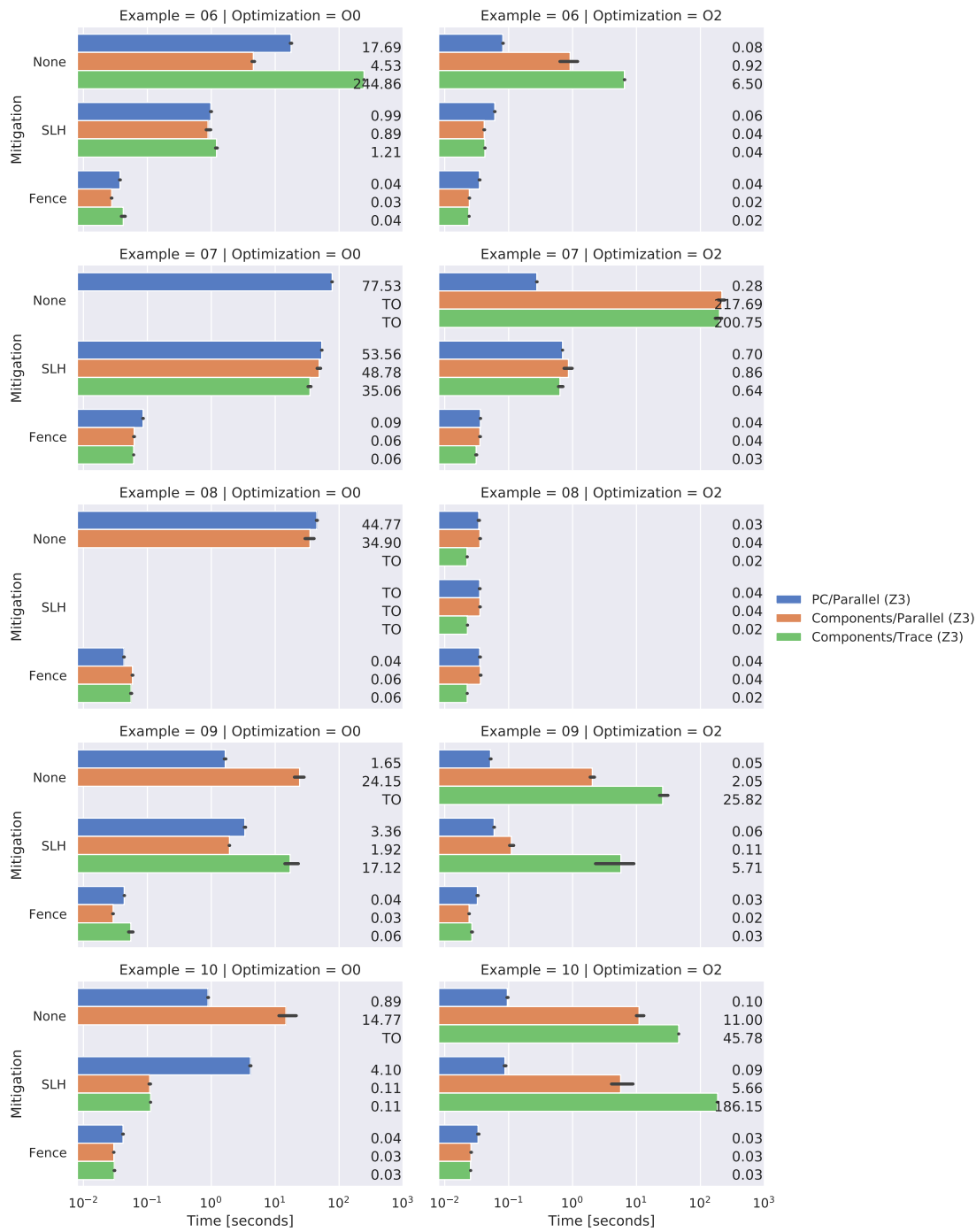


Figure 6.5: Execution Time of SpecBMC using different Settings (Examples 6-10)

6. EVALUATION

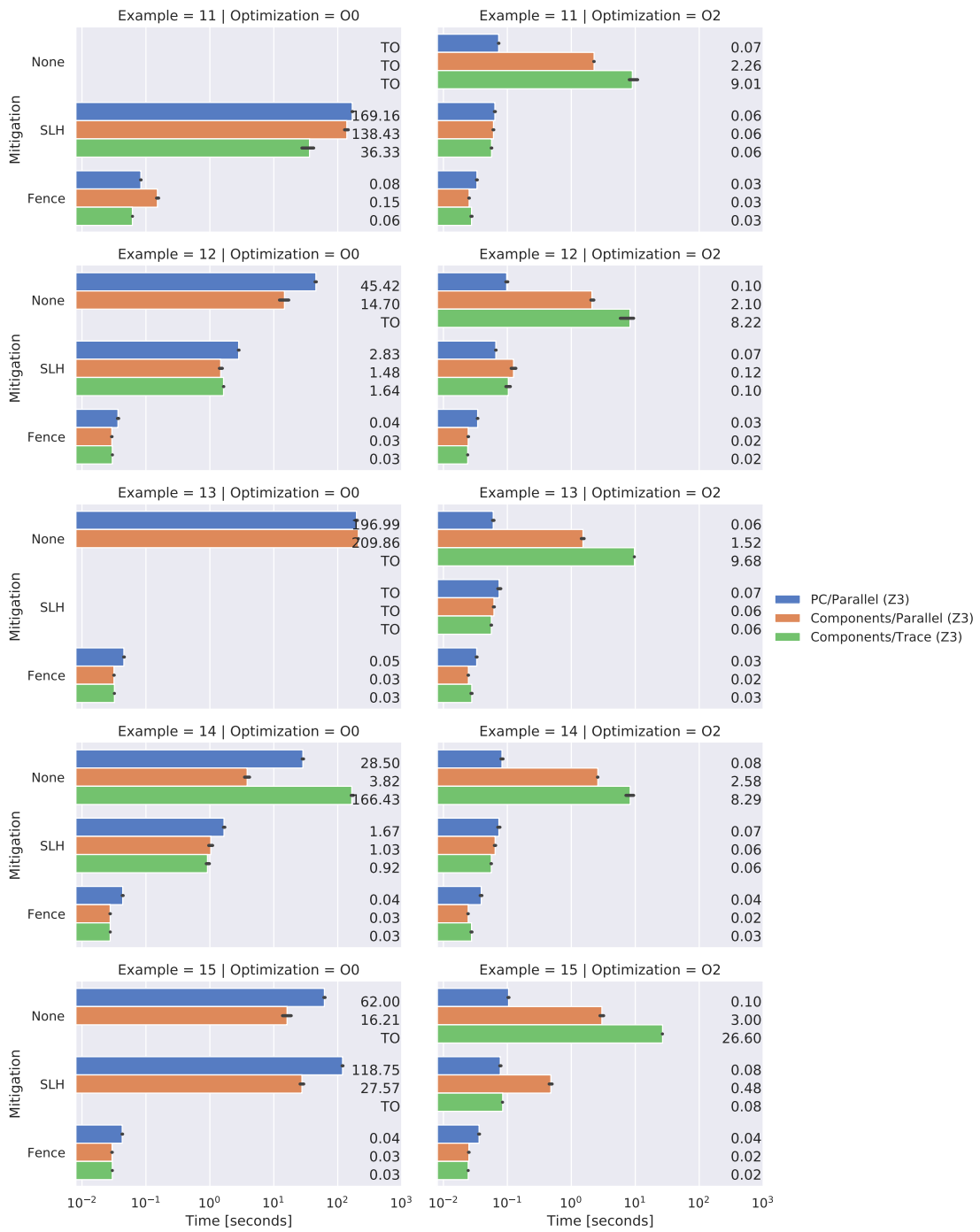


Figure 6.6: Execution Time of SpecBMC using different Settings (Examples 11-15)

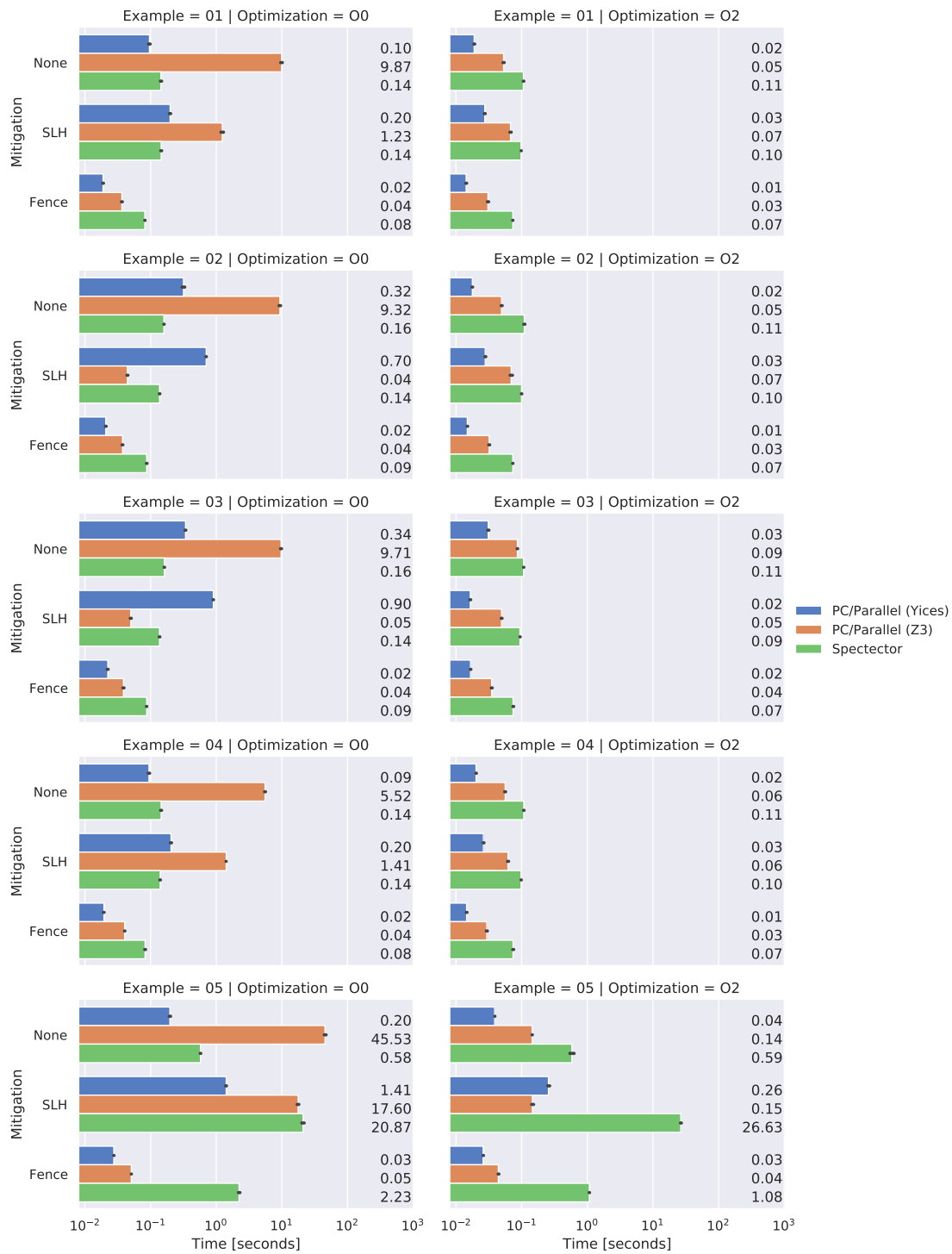


Figure 6.7: Execution Time of SpecBMC and Spectector (Examples 1-5)

6. EVALUATION

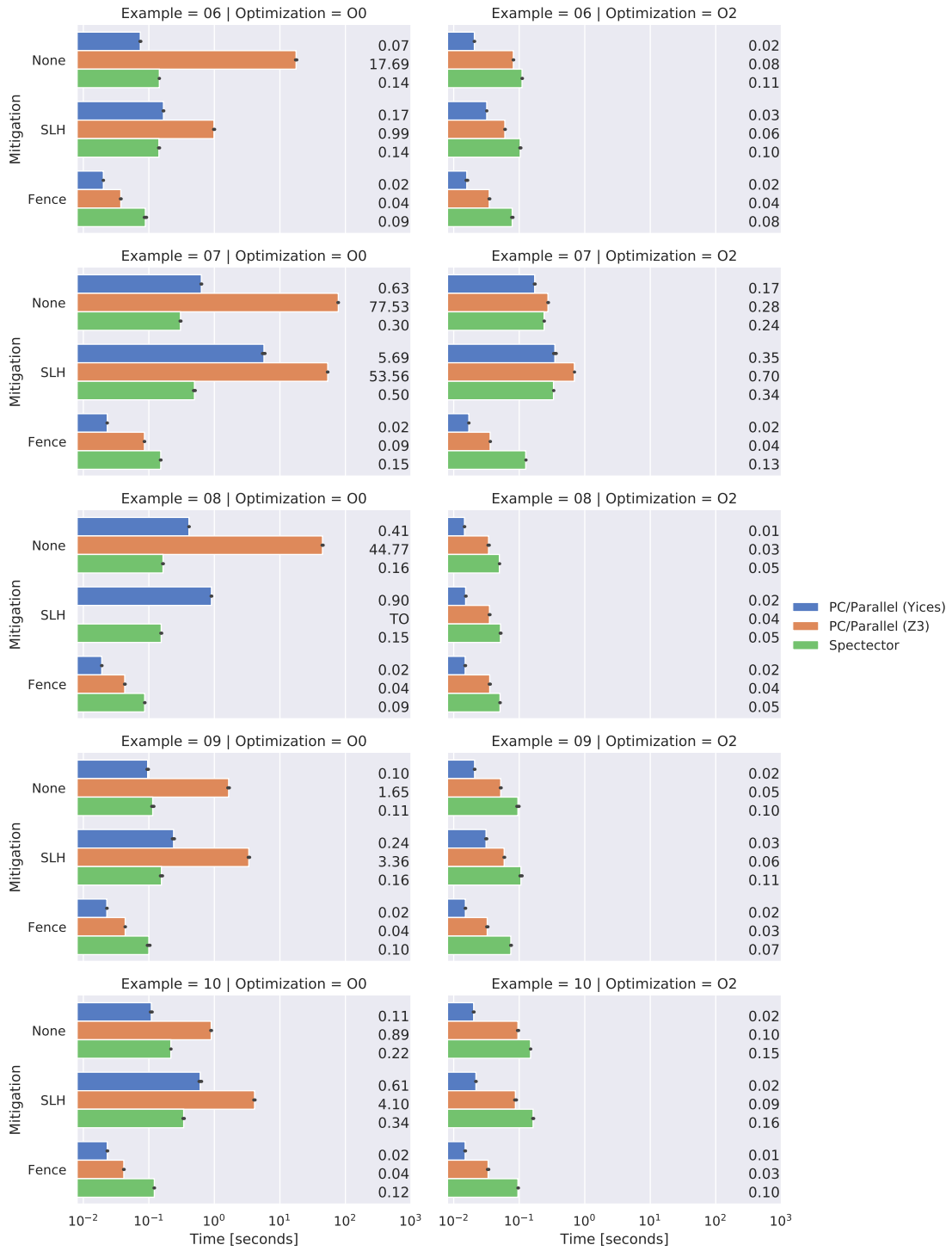


Figure 6.8: Execution Time of SpecBMC and Spectector (Examples 6-10)

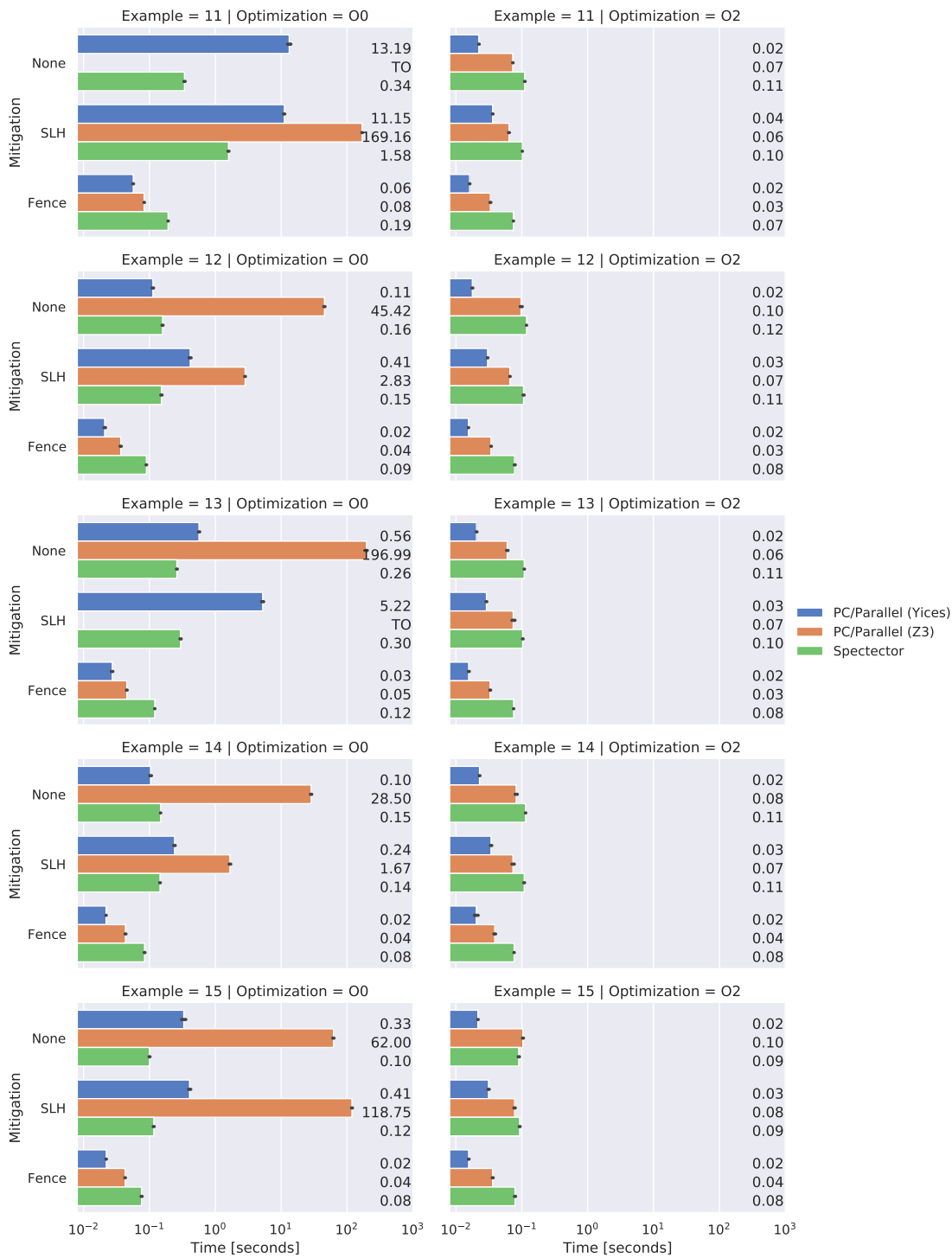


Figure 6.9: Execution Time of SpecBMC and Spectector (Examples 11-15)

6.4 Case Study

Based on the motivating example from Section 1.1.1, we want to check if SpecBMC is able to find the Spectre-PHT vulnerability in the incorrectly backported Linux ptrace patch and show if the proposed fix actually mitigates the vulnerability.

This case study revealed some interesting challenges:

- We had to deal with multiple different less commonly used instructions, such as `rep movsd`. The lifting of the whole Linux kernel revealed many similar instructions. Fortunately, Falcon already has great support for such instructions, so only the translation from Falcon IL to HIR as well as the loop unwinding have to be extended. Additionally, many interesting control-flow specialties have been uncovered, such as directly jumping to a function instead of calling it. This required some improvements in the call reconstruction.
- The most challenging part was the sheer size of the Linux kernel, which caused that SpecBMC almost immediately ran out of memory when lifting the full kernel. The memory consumption has greatly been improved by tweaking the memory layout of SpecBMC's internals as well as using a tiny kernel instead of a full-featured distribution kernel.
- Many additional features were required in SpecBMC to properly set up the initial context of the ptrace functions, such as the possibility to set initial memory and register content as well as the initial state of the CPU flag register.
- Last but not least, the huge size of the resulting control-flow graphs puts a lot of pressure on SpecBMC's internals as well as the back-end SMT solver. When checking huge functions, the solving may take hours instead of just minutes as in the previous tests. To improve this situation, we added the possibility to specify the unwinding bound for individual loops in addition to the global unwinding bound. Additionally, we allow to disable inlining for specific functions, thereby sacrificing analysis soundness. But this is necessary to avoid that the CFG explodes when inlining all the locking, scheduling and other deeply nested functions.

6.4.1 Preparation

We use the source code of Linux 5.8.12 downloaded from kernel.org. A minimal kernel is produced using `make tinyconfig` with the x64 mode enabled. The reason for using a tiny kernel instead of a distribution kernel is simple the memory consumption of SpecBMC, because loading a full featured distribution kernel 5.8.12 from Arch Linux requires almost 28 GB of memory just for lifting and translating the whole kernel into HIR, whereas the peek memory consumption for the tiny kernel is about 1.6 GB. Additionally, we disable inlining for the `ptrace_get_debugreg()` and `ptrace_set_debugreg()` functions, so that we can use them as program entry points in SpecBMC. The kernel is compiled using the GCC compiler. We add `-fno-jump-tables` as additional compiler flag to `KCFLAGS`, so that switch statements are translated into a sequence of branches

instead of a jump table. The reason for not using jump tables is simply that SpecBMC lacks support for indirect jumps, therefore programs using jump tables cannot be checked properly.

6.4.2 Environment

For this case study SpecBMC is invoked with the environment defined in Listing 6.16. It's assumed that all registers are low security and all memory content is high security by default. As seen Listings 6.17 and 6.18, the first argument of the ptrace functions is a pointer to a task struct. We assume that the pointer and the content of the task struct is known to the adversary. By x86-64 calling convention we have that the first argument is passed via the `rdi` register, meaning that `rdi` initially contains the pointer to the task struct. From `/sys/kernel/slab/task_struct/object_size` we get that the task struct has 7936 bytes in size. Therefore, we set the initial value of register `rdi` to `0xffff0000` and require that memory content from address `0xffff0000` to address `0xffff1f00` is of low security.

The assembly code of the inlined `ptrace_set_breakpoint_addr()` function contains a `rep movsd dword [rdi], [rsi]` instruction, which repeatedly copies a double word from memory address `rsi` to memory address `rdi` [24]. In this case the copying is repeated exactly 30 times as the `ecx` register is initialized with `0x1e`. Internally this instruction decrements a counter and automatically adds the offset the the source and target pointer. Based on the direction flag (DF), the instruction can either automatically decrement or increment the addresses. As this instruction is translated into a loop in HIR, we require that this loop, identified by `0x1a`, is unrolled 30 times instead of only 3 times as the other loops. As the DF flag is cleared by default, we initialize it with `false`.

```

1 analysis:
2   spectre_pht: true
3   spectre_stl: false
4   check: only_transient_leaks
5   model: pc
6   unwind: 3
7   unwind_loop:
8     0x1a: 30    # rep movsd
9   inline_ignore:
10    - "register_user_hw_breakpoint"
11    - "modify_user_hw_breakpoint"
12    - "ptrace_register_breakpoint"
13    - "ptrace_write_dr7"
14 architecture:
15   cache: true
16   btb: false
17   pht: true
18 policy:
19   registers:
20     default: low

```

```

21     memory:
22         default: high
23         low:
24             - # child task (see rdi) with size 7936 bytes
25               start: 0xffff0000
26               end: 0xffff1f00
27     setup:
28         init_stack: true
29         registers:
30             rdi: 0xffff0000 # struct task_struct *child
31         flags:
32             DF: false # the direction flag is cleared by default

```

Listing 6.16: SpecBMC Environment for Linux Kernel Case Study

6.4.3 Checking ptrace_get_debugreg()

The current version of `ptrace_get_debugreg()` is shown in Listing 6.17. To check if SpecBMC would have detected the backporting error explained in Section 1.1.1, we reconstruct the problematic change by moving the index masking at line 7 after the memory load at line 8, thereby disabling the load hardening. When checking the incorrectly backported fix of `ptrace_get_debugreg()`, we get a counterexample after about 9.7 seconds. SpecBMC detects that the unprotected memory load at line 8 can be controlled by the adversary and therefore read arbitrary values during transient execution. The secret value can be encoded using one of the two possible covert channels, either via the branch at line 10 or via the memory load at line 11. Running SpecBMC on the current mitigated version of `ptrace_get_debugreg()` shows that implementation in Linux 5.8.12 is secure. The successful check takes about 9.3 seconds.

```

1  unsigned long ptrace_get_debugreg(struct task_struct *tsk, int n)
2  {
3      struct thread_struct *thread = &tsk->thread;
4      unsigned long val = 0;
5
6      if (n < HBP_NUM) {
7          int index = array_index_nospec(n, HBP_NUM);
8          struct perf_event *bp = thread->ptrace_bps[index];
9
10         if (bp)
11             val = bp->hw.info.address;
12     } else if (n == 6) {
13         val = thread->debugreg6;
14     } else if (n == 7) {
15         val = thread->ptrace_dr7;
16     }
17     return val;
18 }

```

Listing 6.17: Linux Kernel `ptrace_get_debugreg()`

6.4.4 Checking `ptrace_set_debugreg()`

While we were running the checks on `ptrace_get_debugreg()` respectively its caller `arch_ptrace()`, SpecBMC detected another similar Spectre-PHT vulnerability in a related function, namely in `ptrace_set_debugreg()`. Compared to the vulnerability in `ptrace_get_debugreg()`, the vulnerability in `ptrace_set_debugreg()` is a little bit trickier as the bounds-check and the memory load are spread over two different functions. The bounds-check is located in `ptrace_set_debugreg()` whereas the memory load and the encode-gadget is located in `ptrace_set_breakpoint_addr()`. Both functions are shown in Listings 6.18 and 6.19.

```

1 int ptrace_set_debugreg(struct task_struct *tsk, int n,
2                       unsigned long val)
3 {
4     struct thread_struct *thread = &tsk->thread;
5     int rc = -EIO;
6
7     if (n < HBP_NUM) {
8         rc = ptrace_set_breakpoint_addr(tsk, n, val);
9     } else if (n == 6) {
10        thread->debugreg6 = val;
11        rc = 0;
12    } else if (n == 7) {
13        rc = ptrace_write_dr7(tsk, val);
14        if (!rc)
15            thread->ptrace_dr7 = val;
16    }
17    return rc;
18 }
```

Listing 6.18: Linux Kernel `ptrace_set_debugreg()`

```

1 int ptrace_set_breakpoint_addr(struct task_struct *tsk, int nr,
2                              unsigned long addr)
3 {
4     struct thread_struct *t = &tsk->thread;
5     struct perf_event *bp = t->ptrace_bps[nr];
6     int err = 0;
7
8     if (!bp) {
9         bp = ptrace_register_breakpoint(tsk,
10            X86_BREAKPOINT_LEN_1, X86_BREAKPOINT_WRITE,
11            addr, true);
12        if (IS_ERR(bp))
13            err = PTR_ERR(bp);
14        else
15            t->ptrace_bps[nr] = bp;
16    } else {
17        struct perf_event_attr attr = bp->attr;
```

```

18
19     attr.bp_addr = addr;
20     err = modify_user_hw_breakpoint(bp, &attr);
21 }
22
23     return err;
24 }

```

Listing 6.19: Linux Kernel ptrace_set_breakpoint_addr()

When checking the `ptrace_set_debugreg()` function as currently implemented in Linux 5.8.12, we get a counterexample after about 626 seconds. SpecBMC detects that the memory load at line 5 of function `ptrace_set_breakpoint_addr()` can be controlled by the adversary and therefore read arbitrary values during transient execution. The adversary speculatively bypasses the bounds-check at line 7 of function `ptrace_set_debugreg()` to reach the memory load during transient execution. The memory load at line 17 of function `ptrace_set_breakpoint_addr()` is then used to encode the secret value stored in `bp` into the cache.

SpecBMC provides the following counterexample for the assembly output shown in Figure 6.10. The index variable `n`, located in register `rsi`, has value `0x40000006`. Transient execution begins at `0xFFFFFFFF810120AB` where the jump is predicted to be not taken. The secret value is loaded at `0xFFFFFFFF810119B9` and later encoded into the cache at `0xFFFFFFFF81011A04`. The encoding of the secret value can be seen at line 16 and 17. SpecBMC detects a second possible covert channel, namely the conditional branch instruction at `0xFFFFFFFF810119C4` allows to leak information if the secret is zero or not.

```

1 0xFFFFFFFF810119B1 rbx = (bvadd rsi 0x6)
2   - A@ rbx = 0x4000000C
3   - B@ rbx = 0x4000000C
4 ...
5 0xFFFFFFFF810119B9 r8 = load(_memory, (bvadd (bvadd 0xFFFF0000 (
   bvmul rbx 0x8)) 0x588))
6   - A# cache_fetch(0x2FFFF05E8, 64)
7   - B# cache_fetch(0x2FFFF05E8, 64)
8   - A@ r8 = 0x1
9   - B@ r8 = 0x40
10 ...
11 0xFFFFFFFF810119F8 rsi = (bvadd r8 0xE0)
12   - A@ rsi = 0xE1
13   - B@ rsi = 0x120
14 ...
15 0xFFFFFFFF81011A04 temp_0xFFFFFFFF81011A04 = load(_memory, rsi)
16   - A# cache_fetch(0xE1, 32)
17   - B# cache_fetch(0x120, 32)
18   - A@ temp_0xFFFFFFFF81011A04 = 0x0
19   - B@ temp_0xFFFFFFFF81011A04 = 0x0
20 0xFFFFFFFF81011A04 _memory = store(_memory, rdi,
   temp_0xFFFFFFFF81011A04)

```

After taking a closer look at the assembly code, the provided counterexample is indeed a possible leak. Via the `sys_ptrace()` system call an adversary is able to reach the unprotected array access in the `ptrace_set_breakpoint_addr()` function, shown in Listing 6.19. The input parameter `nr`, which is used as the array index at line 5, is controllable from userspace. An adversary could cause a speculative out-of-bounds read and thereby load secret information into `bp` during transient execution. By using `bp` as a load address in a successive load, e.g., the load at line 17, the secret information stored in `bp` can be encoded into the cache. The call chain is as follows: (i) `ptrace` system call (ii) `arch_ptrace` with `POKEUSR` request and adversary-chosen address (iii) `ptrace_set_debugreg` with chosen address in argument `n` (iv) `ptrace_set_breakpoint_addr` with chosen address in argument `nr`.

The Spectre-PHT vulnerability can be mitigated by masking the value of `n` in function `ptrace_set_debugreg()` before calling `ptrace_set_breakpoint_addr()`. The mitigated version is shown in Listing 6.20. Running SpecBMC on the mitigated version of `ptrace_set_debugreg()` shows that implementation is secure. The successful check takes about 858 seconds.

```

1 int ptrace_set_debugreg(struct task_struct *tsk, int n,
2                       unsigned long val)
3 {
4     struct thread_struct *thread = &tsk->thread;
5     int rc = -EIO;
6
7     if (n < HBP_NUM) {
8         int index = array_index_nospec(n, HBP_NUM);
9         rc = ptrace_set_breakpoint_addr(tsk, index, val);
10    } else if (n == 6) {
11        thread->debugreg6 = val;
12        rc = 0;
13    } else if (n == 7) {
14        rc = ptrace_write_dr7(tsk, val);
15        if (!rc)
16            thread->ptrace_dr7 = val;
17    }
18    return rc;
19 }
```

Listing 6.20: Mitigated `ptrace_set_debugreg()`

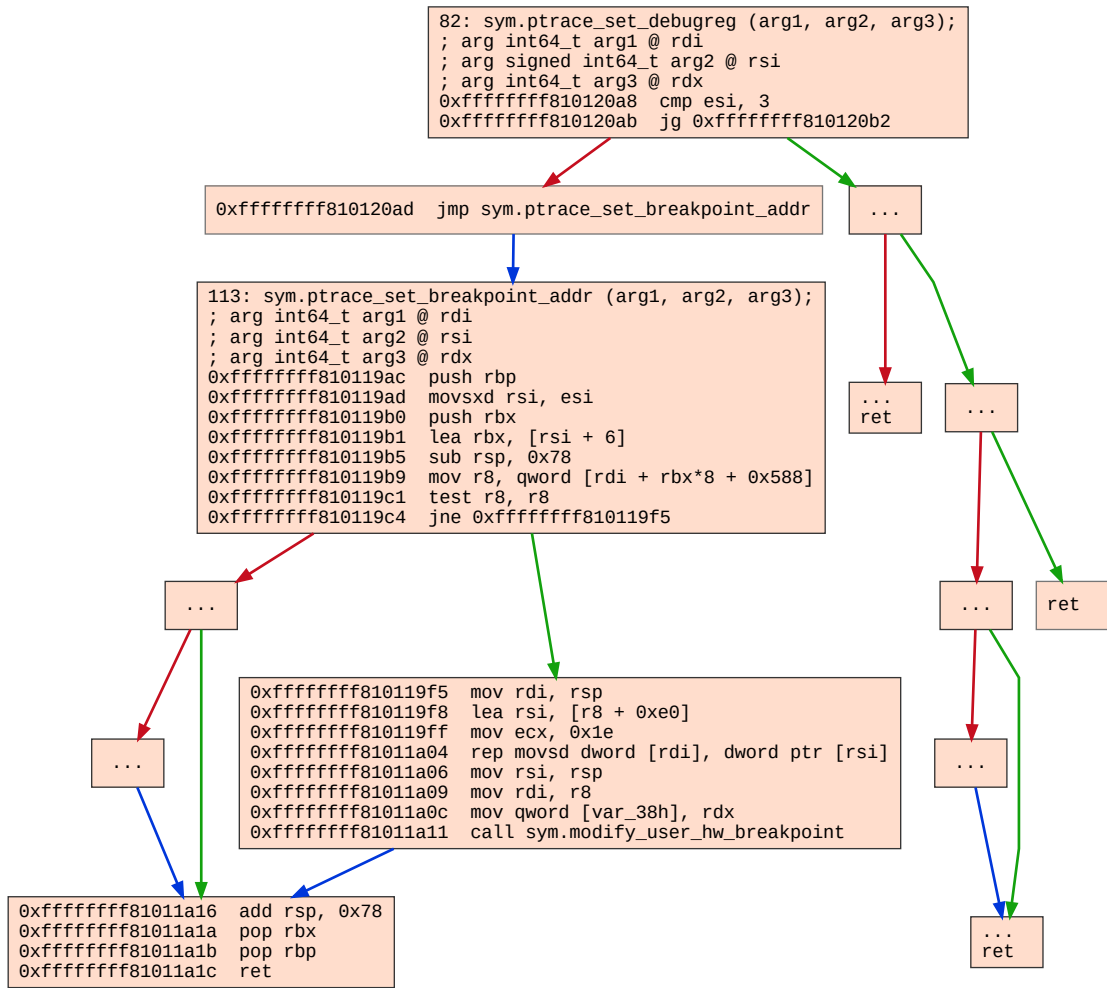


Figure 6.10: Simplified Assembly Code of ptrace_set_debugreg ()

Related Work

7.1 Spectector

```

1 c <- x < array1_size
2 beqz c, 5
3 load y, array1 + x
4 load z, array2 + y * 512

```

Listing 7.1: Spectre-PHT Information Leak

Spectector is a tool for automatic detection of speculative information flows in binary programs [21]. It can detect speculative data as well as control-flow leaks. Spectector uses symbolic execution [5] to derive a set of symbolic traces for a given program. A symbolic trace consists of a sequence of memory and program-counter observations, denoted by `LOAD`, `STORE` and `PC`, as well as path constraints. The transient execution behavior is captured by so-called speculative transactions [21], delimited by `start` and `rollback`. For checking speculative non-interference (SNI) [21], each enumerated symbolic trace is duplicated by means of self-composition, encoded and finally passed on to an SMT solver. If the SMT solver finds a satisfying assignment, then the program is considered insecure and the satisfying assignment is a witness of a transient execution leak.

For example, consider the program shown in Listing 7.1. Spectector derives two symbolic traces for this program. One of these symbolic traces, namely the trace which transiently executes the load instructions, is depicted in Figure 7.1a¹. The transient execution is visualized by red dashed arrows and the non-speculative execution by red solid arrows. The green path shows the speculative transaction beginning at `start(0)` and ending at `rollback(0)`. To check for SNI, the symbolic trace π shown in Figure 7.1a needs to be duplicated. Figure 7.1b shows the self-composition of the symbolic traces π and π^d , symbols which are equal in both traces have been simplified. By asking the SMT solver

¹Generated with <https://github.com/emmanuel099/specgraph>

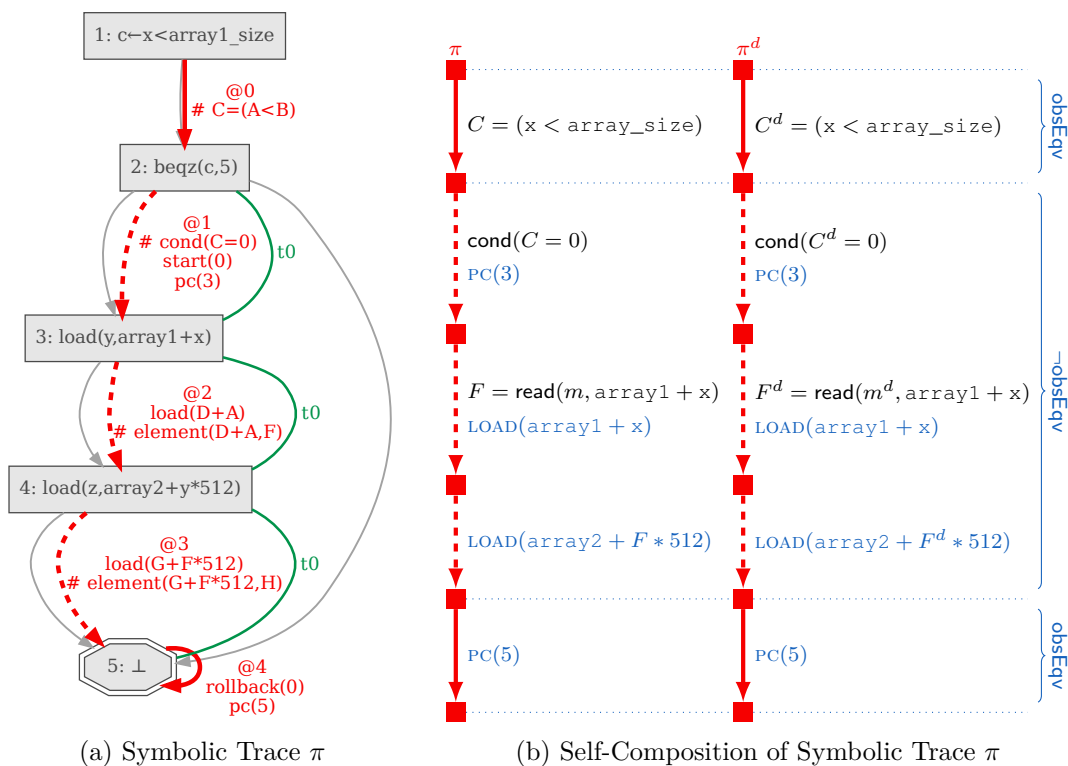


Figure 7.1: Spectre-PHT Information Leak in Spectector

for a satisfying assignment, such that both traces are observable equivalent during non-speculative execution but **not** observable equivalent during transient execution, one can obtain a satisfying assignment with distinct values for F and F^d . The distinct values for F and F^d result in different LOAD observations and therefore the symbolic trace shown in Figure 7.1a and consequently the whole program violates SNI. In other words, the example contains a speculative data leak.

As our work is based on the ideas of Spectector, our definition of SNI as well as our semantics definition is in parts similar to Spectector’s definitions. The three major differences are: (i) Spectector assumes a fixed adversary model [21], inspired by the program-counter security model from Molnar et al. [37], where the adversary can observe the program counter as well as the locations of memory loads and stores. Therefore, Spectector completely abstracts away the microarchitectural components and instead defines a sequence of memory and program-counter observations. In contrast to this, our adversary model allows to select the microarchitectural components which are available for encoding secret information. Therefore, our work introduces high-level definitions of different microarchitectural components, allowing to selectively check for speculative data and control-flow leaks depending on the actually available microarchitectural components. (ii) The transient execution semantics of Spectector defines nested speculative transactions [21], meaning that within a running speculative transaction additional spec-

ulative transactions can be initiated. This may result in long and partially duplicated traces. As we show in Appendix A.3, a single speculative transaction is equally powerful in detecting transient execution bugs. Therefore, our transient execution semantics works on a single speculative state, resulting in simpler traces. (iii) As Spectector is based on symbolic execution, it suffers from the well-known path explosion problem. Dealing with path explosion and loops is one of the major challenges in symbolic execution [5]. Spectector’s approach to mitigate this problem is to limit the path length, thereby forfeiting the soundness of the analysis [21]. Additionally, Spectector requires many SMT queries for each enumerated path, especially when checking for control-flow leaks. In contrast to this, our work is based on BMC. This enables us to directly check all program paths within the SAT/SMT solver [26], thereby allowing the solver to reuse accumulated information across multiple program paths. BMC tackles the state explosion problem by searching for counterexamples of bounded length [12], hence forfeiting analysis soundness similar to Spectector. Nonetheless, BMC provides a strong technique for catching software bugs [26], or in our case transient execution bugs.

```

1  uint8_t arr[256 * 512];
2  uint8_t size = 10;
3
4  void victim() {
5      if (secret == 0) {
6          x = 42;
7      } else {
8          x = 21;
9      }
10     if (x < size) {
11         tmp = arr[x * 512];
12     }
13 }

```

Listing 7.2: Artificial Example showing Differences between Spectector and Our Work

One big difference between Spectector and our approach is that Spectector enforces the exact same (non-speculative) control-flow in both programs, while our approach only requires the same (non-speculative) observations. In contrast to the control-flow restriction, the observations restriction depends on the adversary’s capabilities. Suppose that the adversary can only observe the cache in Listing 7.2. We have at line 10 that the value of x can either be 21 or 42 depending on the value of `secret`. Without speculative execution the adversary is unable to learn anything about `secret`, because line 11 is unreachable for both possible values of x and therefore no `secret`-dependent load occurs. With speculative execution the adversary can learn some information about the `secret`, namely if the `secret` value equals 0 or not. This is possible because the processor will start to speculate at line 10 if `size` isn’t available from cache. Proper training of the branch predictor and flushing `size` from cache will make the `secret`-dependent memory load at line 11 reachable during transient execution. Although Spectector has an option to only detect speculation leaks caused by memory operations, it misses this speculative leak because it forces that all path conditions of both traces match, meaning that for

both traces either $\text{secret} = 0 \wedge \text{secret}^d = 0$ or $\text{secret} \neq 0 \wedge \text{secret}^d \neq 0$ holds, which results in identical values for x and x^d .

7.2 A Formal Approach to Secure Speculation

The work of Cheang et al. [10] introduces a formal approach for secure speculation, consisting of a speculative semantics definition as well as a hyperproperty called trace property-dependent observational determinism (TPOD). Their approach captures transient execution leaks as well as leaks happening during non-speculative execution. A proof-of-concept implementation based on the UCLID5 model checker has been implemented by the authors.

	TPOD	SNI
(i)	$\forall \pi_1, \pi_2, \pi_3, \pi_4.$	$\forall \tilde{\pi}_3, \tilde{\pi}_4.$
(ii)	$\text{conformant}(\pi_1)$ $\wedge \text{conformant}(\pi_2)$ $\wedge \text{conformant}(\pi_3)$ $\wedge \text{conformant}(\pi_4)$	
(iii)	$\wedge \forall i. \neg \text{mispred}(\pi_1^i.n, \pi_1^i.\beta, \pi_1^i.pc)$ $\wedge \exists i. \text{mispred}(\pi_3^i.n, \pi_3^i.\beta, \pi_3^i.pc)$ $\wedge \text{op}_{\mathcal{H}}(\pi_1) = \text{op}_{\mathcal{H}}(\pi_3)$	$\pi_1 = \text{ns}(\tilde{\pi}_3)$
(iv)	$\wedge \forall i. \neg \text{mispred}(\pi_2^i.n, \pi_2^i.\beta, \pi_2^i.pc)$ $\wedge \exists i. \text{mispred}(\pi_4^i.n, \pi_4^i.\beta, \pi_4^i.pc)$ $\wedge \text{op}_{\mathcal{H}}(\pi_2) = \text{op}_{\mathcal{H}}(\pi_4)$	$\wedge \pi_2 = \text{ns}(\tilde{\pi}_4)$
(v)	$\wedge \pi_3^0 \approx_{\mathcal{L}} \pi_4^0$	$\wedge \tilde{\pi}_3^0 \sim_{\mathcal{L}} \tilde{\pi}_4^0$
(vi)	$\wedge \text{op}_{\mathcal{L}}(\pi_1) = \text{op}_{\mathcal{L}}(\pi_2) = \text{op}_{\mathcal{L}}(\pi_3) = \text{op}_{\mathcal{L}}(\pi_4)$	
(vii)	$\wedge \pi_1 \approx_{\mathcal{L}} \pi_2$	$\wedge \pi_1 \approx_{\text{Obs}} \pi_2$
(viii)	$\implies \pi_3 \approx_{\mathcal{L}} \pi_4$	$\implies \tilde{\pi}_3 \approx_{\text{Obs}} \tilde{\pi}_4$

Table 7.1: Comparison of TPOD and SNI Hyperproperty

Table 7.1 shows how our definition of SNI introduced in Section 3.5 relates to TPOD. A rough comparison of both definitions: (i) As TPOD is defined as a 4-safety hyperproperty it requires four execution traces, whereas SNI requires only two speculative execution traces. Meaning that our definition can be checked using only 2-way self-composition, which is likely to be more scalable than their approach. (ii) The conformant predicate constrains the traces to valid executions only, required to constrain the adversary component. As this is an implementation detail of TPOD our definition of SNI has no counterpart. (iii) TPOD constrains the traces π_1 and π_3 such that both non-spec-

ulatively execute the same instructions with low-equivalent memory. Additionally, it allows transient execution only for trace π_3 . This corresponds roughly to our definition, where we obtain the non-speculative trace π_1 by simply removing all speculative configurations from the speculative trace $\tilde{\pi}_3$. The difference is that the SNI traces π_1 and $\tilde{\pi}_3$ have low- and high-equivalent memory in all non-speculative configurations, whereas the TPOD traces π_1 and π_3 have only low-equivalent memory in all non-speculative configurations. (iv) Same as before but with traces π_2 and π_4 . (v) Both definitions require that the initial states are low-equivalent, meaning that low-security registers and memory locations agree on the same values. (vi) TPOD enforces that the adversary component executes the same instructions in all traces. As this is an implementation detail of TPOD our definition of SNI has no counterpart. (vii) TPOD requires that the non-speculative traces π_1 and π_2 are low-equivalent, where low-equivalence includes the microarchitectural components as well as low-security registers and memory locations. Compared to this, SNI requires that the non-speculative traces π_1 and π_2 have the same observational effects on the microarchitectural components, meaning that in TPOD an adversary can observe registers and memory in addition to the microarchitectural covert channel, whereas in SNI only the microarchitectural covert channel is visible to the attacker. (viii) In TPOD a program is considered secure if the speculative traces π_3 and π_4 are low-equivalent. As before, low-equivalence includes the microarchitectural components as well as low-security registers and memory locations. Therefore, TPOD will also detect non-speculative memory leaks. In SNI a program is considered secure if an attacker cannot distinguish between the speculative traces $\tilde{\pi}_3$ and $\tilde{\pi}_4$ by observing their effects on the microarchitectural components.

In brief summary: TPOD detects non-speculative leaks in addition to transient execution leaks [10], including memory leaks, at the cost of a 4-safety hyperproperty, whereas SNI defines a less expensive 2-safety hyperproperty but can only detect transient execution leaks. Note that our approach would also be able to detect non-speculative leaks in addition to transient execution leaks when removing assumption (vii) $\pi_1 \approx_{\text{Obs}} \pi_2$ which requires that the non-speculative observations are indistinguishable.

7.3 SCADET

SCADET is a program analysis tool for detecting different variants of Prime+Probe side-channel attacks, targeting the L1 as well as the last-level cache [44]. Because Prime+Probe allows an adversary to reconstruct secrets from cache-based covert channels, this tool is also able to detect some sorts of Spectre-style attacks. SCADET uses dynamic binary instrumentation to inject monitoring code for collecting memory access traces while the program is running. As Prime+Probe attacks require many consecutive memory accesses to fill the cache as well as to determine the set of replaced memory locations [39, 50], such attacks have conspicuous memory access patterns. Therefore, offline analysis is used to search for Prime+Probe patterns in the collected traces. According to the authors of SCADET, their technique achieves an TPR of 100% and an average FPR of about 7.4% in detecting Prime+Probe attacks [44].

7.4 Spectre is here to stay

The work of Mcilroy et al. [33] introduces a formal model for analyzing and reasoning about side-channels in the presence of speculative execution. Similar to our work, they introduce state transition systems for both the architectural and microarchitectural view of the processor. Their microarchitectural model includes a reorder buffer to keep track of uncommitted instructions, both for out-of-order evaluation and branch prediction. Furthermore, they define a timer model and explicitly formalize the timing behavior of the different instructions in interplay with the microarchitectural components. Based on this model they show that despite on constant-time implementations, full mitigation of timing channels is impossible because of the various optimizations in place [33]. Additionally, they give a simulation relation from the microarchitectural to the architectural STS, showing that each architectural state has one or more corresponding microarchitectural states. They conclude that vulnerabilities resulting from speculative execution are fundamental design flaws arising from the mapping of one architectural state to multiple concrete microarchitectural states [33], therefore inevitable affecting all processors that perform speculation.

7.5 CacheAudit

CacheAudit is a static, quantitative analysis tool for exploring cache-based side-channels in binary programs [14]. In CacheAudit a program is considered secure if the observable cache states are independent from any secret input. Secret-dependent cache states allow an adversary to partially or completely reconstruct the secrets (i) by determining the set of cached memory blocks upon termination of the program (ii) by observing traces of cache hits and misses during execution of the program (iii) or by measuring the program's execution time depending on the memory access delays. In CacheAudit these three different views are denoted as access-based, trace-based, respectively timing-based adversary. As the interaction of the program with the cache heavily depends on the actual type of the cache, CacheAudit defines a sophisticated cache model with multiple different replacement strategies to increase its analysis precision.

In contrast to our work which focuses solely on non-interference, CacheAudit gives quantitative security guarantees [14], meaning that it can give an upper bound of how much information is actually leaked by the program. Additionally, our cache model defined in Section 4.1 is rather simplistic compared to CacheAudit and therefore less accurate, but extending our model with different replacement strategies similar to CacheAudit should easily be doable. Because CacheAudit doesn't model speculative execution it can only detect non-speculative leaks in contrast to our work.

7.6 SpecFuzz

SpecFuzz is a dynamic testing tool to uncover speculative execution vulnerabilities by means of fuzzing [38]. At compile-time the software under test is instrumented with

speculative execution simulation and integrity checks, such as absence of invalid memory accesses. Speculative execution simulation, currently limited to Spectre-PHT only, forcefully executes code paths which would otherwise only be reached when a conditional branch is mis-predicted on the CPU. This is achieved by adding additional control-flow edges into the control-flow graph to cover the mis-predicted code paths.

After compilation the instrumented binary is then examined by a fuzzing tool. The fuzzing tool repeatedly executes the program with random inputs to stress various different code paths. If the fuzzing process triggers an invalid memory access error in the instrumented binary, a speculative execution vulnerability has been detected. The authors of SpecFuzz evaluated their tool based on the well-known Kocher examples [27] and some example applications, such as the OpenSSL library, the Brotli compression library or the LibYAML parsing library. SpecFuzz successfully detected all 15 Kocher vulnerabilities [38] and uncovered a tremendous amount of speculative execution vulnerabilities in the example applications.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Conclusion

First, we define a formal semantics for transient execution which covers the speculative behavior of the instructions, the branch predictor as well as the memory disambiguator. Furthermore, we formalize microarchitectural components such as the cache or the branch target buffer to capture the microarchitectural side effects of the instructions. The microarchitectural components allow us to model cache- as well as branch-predictor-based covert channels. A speculative non-interference hyperproperty is defined, which allows to reason about the security of a program in respect to transient execution attacks. A program is considered secure, if the program executed on a CPU with speculation has the same adversary observable security sensitive side effects as if the program is executed on a CPU without speculation.

Based on our definition of SNI, a bounded software model checker named SpecBMC is implemented. SpecBMC can automatically detect Spectre-style vulnerabilities in binary programs. We use self-composition to transform the SNI hyperproperty into a safety problem which can then be checked by off-the-shelf SMT solvers like Z3 or Yices2. The tool is released under the free and open source Apache-2.0 license and can be downloaded from <https://github.com/emmanuel099/specbmc>.

Finally, we validate SpecBMC against different known Spectre-PHT/STL vulnerabilities and proposed mitigation approaches. Furthermore, we evaluate SpecBMC on the Kocher examples and show that SpecBMC detects the same leaks as Spectector [21] in all the 15 examples. By comparing the solving time of different observation models, we show that the parallel observation model is less expensive than the trace observation model. Additionally, we show that the components model has a measurable overhead compared to the simpler program counter model. From this we conclude that our abstraction level may be too low and instead the side effects should be compared directly without first encoding them into the microarchitectural state.

To complete this thesis, we conduct a small case study where we check the Linux kernel backporting error from our motivating example in Section 1.1. We show that SpecBMC can be used to check sizeable binaries and that the backporting bug could

have been detected with static analysis tools like SpecBMC. Our case study revealed a similar yet unpatched Spectre-PHT vulnerability in a related function of the Linux kernel's ptrace module.

The main contributions of this thesis are an extensible transient execution semantics and a static binary analysis tool that detects Spectre-PHT and Spectre-STL vulnerabilities in binary programs.

8.1 Future Work

While working on this thesis we identified some limitations and ideas for improvements:

- The transient execution semantics as well as the implementation lacks support for indirect branches. As we have seen in our case study, indirect jumps are almost unavoidable when working with real-word binaries. Indirect jumps are also necessary when checking for Spectre-BTB and Spectre-RSB vulnerabilities, which we ignored in this thesis.
- SpecBMC relies on the self-composition approach to transform the SNI hyper-property into a safety problem. The current implementation uses eager self-composition, meaning that we reason on two full copies of a program. The full duplication can become quite expensive. Some promising techniques for more efficient self-composition have been proposed in literature, such as lazy self-composition [51]. The lazy self-composition approach uses symbolic taint analysis to only duplicate the relevant parts of the program, while sharing the rest of the program in both executions. We assume that such approaches would be well-suited for our use case, as we basically restrict the non-speculation part of the program to be indistinguishable for the adversary, meaning that many variables are basically forced to be untainted. Changing from eager to lazy self-composition should be relatively straightforward in the MIR to LIR lowering step.
- As shown in Section 5.4.1, counterexamples are currently exported as annotated control-flow graphs. While this visualization works great for small programs, it becomes inconvenient when the programs get larger and larger. For large counterexamples the identification of the relevant side effects is relatively time consuming and confusing. Automatically extracting the relevant parts of the counterexample and giving a textual explanation of where the leak occurred and from where the leaked secret values originate from would be a fantastic enhancement.
- SpecBMC is currently limited to single queries, meaning that once a leak is detected the check is terminated. Incremental solving would allow SpecBMC to continue after the first leak, meaning that multiple leaks could be detected in a single run. As incremental solving allows the SMT solver to retain the state, we expect that further leaks are detected relatively fast compared to the first leak. Incremental solving requires some extensions in how SpecBMC interacts with the solver and additionally it should be considered how already found leaks can efficiently be “ignored” in the solver.

Proofs

A.1 Trace obtained by $\text{ns}(\pi)$ is valid

Let $\tilde{\pi}_s$ be a valid speculative trace. Let $\tilde{\pi}_{ns}$ be the corresponding non-speculative trace obtained by $\text{ns}(\tilde{\pi}_s)$, then $\tilde{\pi}_{ns}$ is a semantically valid trace.

1. **Barrier instruction:** Let ρ be a program where $\rho(pc) = \text{spbarr}$. As the barrier instruction cannot start a new speculative execution, we get the following simple speculative trace:

$$\tilde{\pi}_s = \langle \varphi, \sigma, pc, \mathcal{Y}, \perp \rangle \xrightarrow{\text{SPECBARRIER-NS}} \langle \varphi, \sigma, pc + 1, \mathcal{Y}, \perp \rangle$$

By removing all speculative intermediate configurations we obtain the following non-speculative trace:

$$\tilde{\pi}_{ns} = \text{ns}(\tilde{\pi}_s) = \langle \varphi, \sigma, pc, \mathcal{Y}, \perp \rangle \xrightarrow{\text{SPECBARRIER-NS}} \langle \varphi, \sigma, pc + 1, \mathcal{Y}, \perp \rangle$$

This corresponds to the non-speculative SPECBARRIER-NS rule.

2. **Observe instruction:** Same proof as for the barrier instruction.
3. **Lifted instructions:** Same proof as for the barrier instruction.
4. **Branch instruction:** Let ρ be a program where $\rho(pc) = \text{beqz } x, \ell$ and let φ be a register assignment where $\varphi(x) = 0$. Assume that the branch instruction starts a new transient execution with speculation window of size k , therefore we have a predictor \mathcal{Y} with $\text{speculate}(\mathcal{Y}, pc)$, $\neg \text{taken}(\mathcal{Y}, pc)$ and $\text{speculation-window}(\mathcal{Y}, pc) = k$. We get the following speculative trace where the transient execution ends after at most k steps:

$$\begin{aligned} \tilde{\pi}_s &= \langle \varphi, \sigma, pc, \mathcal{Y}, \perp \rangle \xrightarrow{\text{SPECBRANCHPRED-NS}} \langle \varphi, \sigma, pc + 1, \mathcal{Y}, (k, \varphi, \sigma, pc) \rangle \\ &\quad \xrightarrow{\text{spec}^j} \langle \varphi', \sigma', pc', \mathcal{Y}, (0, \varphi, \sigma, pc) \rangle \xrightarrow{\text{SPECROLLBACK}} \langle \varphi, \sigma, \ell, \mathcal{Y}, \perp \rangle \\ &\quad \text{where } 1 \leq j \leq k \end{aligned}$$

By removing all speculative intermediate configurations we obtain the following non-speculative trace:

$$\tilde{\pi}_{ns} = \text{ns}(\tilde{\pi}_s) = \langle \varphi, \sigma, pc, \mathcal{Y}, \perp \rangle \xrightarrow{\text{spec}} \langle \varphi, \sigma, \ell, \mathcal{Y}, \perp \rangle$$

This corresponds to the non-speculative SPECBRANCH-NS rule. For $\varphi(x) \neq 0$ the proof is similar.

5. **Store instruction:** Let ρ be a program where $\rho(pc) = \text{store } x, e$. Assume that the store instruction starts a new transient execution with speculation window of size k , therefore we have a predictor \mathcal{Y} with $\text{speculate}(\mathcal{Y}, pc)$ and $\text{speculation-window}(\mathcal{Y}, pc) = k$. We get the following speculative trace where the transient execution ends after at most k steps:

$$\begin{aligned} \tilde{\pi}_s &= \langle \varphi, \sigma, pc, \mathcal{Y}, \perp \rangle \xrightarrow{\text{SPECSTOREBYPASS-NS}} \langle \varphi, \sigma, pc + 1, \mathcal{Y}, (k, \varphi, \sigma, pc) \rangle \\ &\xrightarrow{\text{spec}^j} \langle \varphi', \sigma', pc', \mathcal{Y}, (0, \varphi, \sigma, pc) \rangle \xrightarrow{\text{SPECROLLBACK}} \langle \varphi, \sigma'', pc + 1, \mathcal{Y}, \perp \rangle \\ &\text{where } 1 \leq j \leq k \text{ and } \sigma'' = \sigma[[e]]\varphi \mapsto \varphi(x) \end{aligned}$$

By removing all speculative intermediate configurations we obtain the following non-speculative trace:

$$\begin{aligned} \tilde{\pi}_{ns} = \text{ns}(\tilde{\pi}_s) &= \langle \varphi, \sigma, pc, \mathcal{Y}, \perp \rangle \xrightarrow{\text{spec}} \langle \varphi, \sigma'', pc + 1, \mathcal{Y}, \perp \rangle \\ &\text{where } \sigma'' = \sigma[[e]]\varphi \mapsto \varphi(x) \end{aligned}$$

This corresponds to the non-speculative SPECSTORE-NS rule.

A.2 Well-definedness of instruction's microarchitectural effects

We show that the instruction's microarchitectural effects are the same in case of correct prediction as well as mis-prediction + rollback.

1. **Branch instruction:** Let ρ be a program where $\rho(pc) = \text{beqz } x, \ell$ and let φ be a register assignment where $\varphi(x) = 0$.

Case A: Assume $\neg \text{speculate}(\mathcal{Y}, pc)$.

$$\begin{aligned} \langle \varphi, \sigma, pc, \mathcal{Y}, \perp, \kappa \rangle &\xrightarrow{\mu\text{ARCH (SPECBRANCH-NS)}} \langle \varphi, \sigma, \ell, \mathcal{Y}, \perp, \kappa' \rangle \\ &\text{where } \kappa' = \mathcal{K}\text{-effects}(\kappa, \varphi, \sigma, pc, \varphi, \sigma, \ell) \end{aligned}$$

Case B: Assume $\text{speculate}(\mathcal{Y}, pc)$ and $\neg\text{taken}(\mathcal{Y}, pc)$ and w.l.o.g. $\text{speculation-window}(\mathcal{Y}, pc) = 0$.

$$\begin{aligned} & \langle \varphi, \sigma, pc, \mathcal{Y}, \perp, \kappa \rangle \\ & \xrightarrow{\mu_{\text{ARCH}}(\text{SPECBRANCHPRED-NS})} \langle \varphi, \sigma, pc + 1, \mathcal{Y}, (0, \varphi, \sigma, pc), \kappa \rangle \\ & \xrightarrow{\mu_{\text{ARCH}}(\text{SPECROLLBACK})} \langle \varphi, \sigma, \ell, \mathcal{Y}, \perp, \kappa'' \rangle \\ & \text{where } \kappa'' = \mathcal{K}\text{-effects}(\kappa, \varphi, \sigma, pc, \varphi, \sigma, \ell) \end{aligned}$$

By transitivity we obtain that $\kappa' = \kappa''$. Therefore, the branch instruction gives the same microarchitectural effects in both cases. For $\varphi(x) \neq 0$ the proof is similar.

2. **Store instruction:** Let ρ be a program where $\rho(pc) = \text{store } x, e$.

Case A: Assume $\neg\text{speculate}(\mathcal{Y}, pc)$.

$$\begin{aligned} & \langle \varphi, \sigma, pc, \mathcal{Y}, \perp, \kappa \rangle \xrightarrow{\mu_{\text{ARCH}}(\text{SPECSTORE-NS})} \langle \varphi, \sigma', pc + 1, \mathcal{Y}, \perp, \kappa' \rangle \\ & \text{where } \sigma' = \sigma[[e]]\varphi \mapsto \varphi(x) \text{ and } \kappa' = \mathcal{K}\text{-effects}(\kappa, \varphi, \sigma, pc, \varphi, \sigma', pc + 1) \end{aligned}$$

Case B: Assume $\text{speculate}(\mathcal{Y}, pc)$ and w.l.o.g. $\text{speculation-window}(\mathcal{Y}, pc) = 0$.

$$\begin{aligned} & \langle \varphi, \sigma, pc, \mathcal{Y}, \perp, \kappa \rangle \\ & \xrightarrow{\mu_{\text{ARCH}}(\text{SPECSTOREBYPASS-NS})} \langle \varphi, \sigma, pc + 1, \mathcal{Y}, (0, \varphi, \sigma, pc), \kappa \rangle \\ & \xrightarrow{\mu_{\text{ARCH}}(\text{SPECROLLBACK})} \langle \varphi, \sigma'', pc + 1, \mathcal{Y}, \perp, \kappa'' \rangle \\ & \text{where } \sigma'' = \sigma[[e]]\varphi \mapsto \varphi(x) \text{ and } \kappa'' = \mathcal{K}\text{-effects}(\kappa, \varphi, \sigma, pc, \varphi, \sigma'', pc + 1) \end{aligned}$$

By transitivity we obtain that $\sigma' = \sigma''$ and consequently also $\kappa' = \kappa''$. Therefore, the store instruction gives the same microarchitectural effects in both cases.

A.3 Single speculative state is enough

We extend the transient execution semantics introduced in Section 3.3 with a stack of speculative states. This allows us to capture nested speculative behavior similar to Spectector [21]. In the following all the semantics rules of `spec` have been adjusted by replacing the single speculative state Δ with a stack of speculative states \mathcal{S} , respectively the empty speculative state \perp with an empty stack denoted as $[]$. Notable differences between `spec+` and `spec` are: (i) `SPEC+BRANCHPRED-S` pushes a new speculative state onto the stack, meaning that during transient execution the resolved branch is executed in addition to the speculated one. (ii) `SPEC+STOREBYPASS-S` pushes a new speculative state onto the stack, meaning that during transient execution the non-bypassed store path is executed in addition to the bypassed one. (iii) Rollback doesn't immediately stop the transient execution but instead only removes the speculative state from the top

of the stack, meaning that the transient execution continues until the stack is empty. If multiple speculative states for the currently resolved \widehat{pc} exist (because of loops), then the outermost speculative state with \widehat{pc} will be used for rollback instead of the top one. This is motivated by the behavior of the re-order buffer. If a prediction for $\rho(pc)$ gets resolved, all corresponding entries depending on the prediction of $\rho(pc)$ are either committed or discarded [47].

$$\begin{array}{c}
\text{SPEC}_+\text{LIFT-NS} \frac{\text{lift-inst}(\rho(pc)) \quad \langle \varphi, \sigma, pc \rangle \longrightarrow \langle \varphi', \sigma', pc' \rangle}{\langle \varphi, \sigma, pc, \mathcal{Y}, [] \rangle \xrightarrow{\text{spec}_+} \langle \varphi', \sigma', pc', \mathcal{Y}, [] \rangle} \\
\text{SPEC}_+\text{LIFT-S} \frac{\text{lift-inst}(\rho(pc)) \quad \langle \varphi, \sigma, pc \rangle \longrightarrow \langle \varphi', \sigma', pc' \rangle \quad \omega > 0}{\langle \varphi, \sigma, pc, \mathcal{Y}, \mathcal{S} \triangleleft (\omega, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \rangle \xrightarrow{\text{spec}_+} \langle \varphi', \sigma', pc', \mathcal{Y}, \mathcal{S} \triangleleft (\omega - 1, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \rangle} \\
\text{SPEC}_+\text{OBSERVE} \frac{\rho(pc) = \text{obs}}{\langle \varphi, \sigma, pc, \mathcal{Y}, \mathcal{S} \rangle \xrightarrow{\text{spec}_+} \langle \varphi, \sigma, pc + 1, \mathcal{Y}, \mathcal{S} \rangle} \\
\text{SPEC}_+\text{BARRIER-NS} \frac{\rho(pc) = \text{spbarr}}{\langle \varphi, \sigma, pc, \mathcal{Y}, [] \rangle \xrightarrow{\text{spec}_+} \langle \varphi, \sigma, pc + 1, \mathcal{Y}, [] \rangle} \\
\text{SPEC}_+\text{BARRIER-S} \frac{\rho(pc) = \text{spbarr} \quad \omega > 0}{\langle \varphi, \sigma, pc, \mathcal{Y}, (\omega, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) :: \mathcal{S} \rangle \xrightarrow{\text{spec}_+} \langle \varphi, \sigma, pc, \mathcal{Y}, [(0, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc})] \rangle} \\
\text{SPEC}_+\text{BRANCH-NS} \frac{\rho(pc) = \text{beqz } x, \ell \quad \neg \text{speculate}(\mathcal{Y}, pc) \quad \langle \varphi, \sigma, pc \rangle \longrightarrow \langle \varphi', \sigma', pc' \rangle}{\langle \varphi, \sigma, pc, \mathcal{Y}, [] \rangle \xrightarrow{\text{spec}_+} \langle \varphi', \sigma', pc', \mathcal{Y}, [] \rangle} \\
\text{SPEC}_+\text{BRANCHPRED-NS} \frac{\begin{array}{c} \rho(pc) = \text{beqz } x, \ell \\ \text{speculate}(\mathcal{Y}, pc) \quad pc' = \text{ite}(\text{taken}(\mathcal{Y}, pc), \ell, pc + 1) \\ \omega' = \text{speculation-window}(\mathcal{Y}, pc) \end{array}}{\langle \varphi, \sigma, pc, \mathcal{Y}, [] \rangle \xrightarrow{\text{spec}_+} \langle \varphi, \sigma, pc', \mathcal{Y}, [(\omega', \varphi, \sigma, pc)] \rangle} \\
\text{SPEC}_+\text{BRANCH-S} \frac{\begin{array}{c} \rho(pc) = \text{beqz } x, \ell \\ \neg \text{speculate}(\mathcal{Y}, pc) \quad \omega > 0 \quad \langle \varphi, \sigma, pc \rangle \longrightarrow \langle \varphi', \sigma', pc' \rangle \end{array}}{\langle \varphi, \sigma, pc, \mathcal{Y}, \mathcal{S} \triangleleft (\omega, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \rangle \xrightarrow{\text{spec}_+} \langle \varphi', \sigma', pc', \mathcal{Y}, \mathcal{S} \triangleleft (\omega - 1, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \rangle} \\
\text{SPEC}_+\text{BRANCHPRED-S} \frac{\begin{array}{c} \rho(pc) = \text{beqz } x, \ell \\ \text{speculate}(\mathcal{Y}, pc) \quad \omega > 0 \quad pc' = \text{ite}(\text{taken}(\mathcal{Y}, pc), \ell, pc + 1) \\ \omega' = \min(\omega - 1, \text{speculation-window}(\mathcal{Y}, pc)) \\ \mathcal{S}' = \mathcal{S} \triangleleft (\omega - 1, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \triangleleft (\omega', \varphi, \sigma, pc) \end{array}}{\langle \varphi, \sigma, pc, \mathcal{Y}, \mathcal{S} \triangleleft (\omega, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \rangle \xrightarrow{\text{spec}_+} \langle \varphi, \sigma, pc', \mathcal{Y}, \mathcal{S}' \rangle}
\end{array}$$

$$\begin{array}{c}
\text{SPEC}_+ \text{STORE-NS} \frac{\rho(pc) = \text{store } x, e \quad \neg \text{speculate}(\mathcal{Y}, pc) \quad \langle \varphi, \sigma, pc \rangle \longrightarrow \langle \varphi', \sigma', pc' \rangle}{\langle \varphi, \sigma, pc, \mathcal{Y}, [] \rangle \xrightarrow{\text{spec}_+} \langle \varphi', \sigma', pc', \mathcal{Y}, [] \rangle} \\
\\
\text{SPEC}_+ \text{STOREBYPASS-NS} \frac{\rho(pc) = \text{store } x, e \quad \text{speculate}(\mathcal{Y}, pc) \quad \omega' = \text{speculation-window}(\mathcal{Y}, pc)}{\langle \varphi, \sigma, pc, \mathcal{Y}, [] \rangle \xrightarrow{\text{spec}_+} \langle \varphi, \sigma, pc + 1, \mathcal{Y}, [(\omega', \varphi, \sigma, pc)] \rangle} \\
\\
\text{SPEC}_+ \text{STORE-S} \frac{\rho(pc) = \text{store } x, e \quad \neg \text{speculate}(\mathcal{Y}, pc) \quad \omega > 0 \quad \langle \varphi, \sigma, pc \rangle \longrightarrow \langle \varphi', \sigma', pc' \rangle}{\langle \varphi, \sigma, pc, \mathcal{Y}, \mathcal{S} \triangleleft (\omega, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \rangle \xrightarrow{\text{spec}_+} \langle \varphi', \sigma', pc', \mathcal{Y}, \mathcal{S} \triangleleft (\omega - 1, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \rangle} \\
\\
\text{SPEC}_+ \text{STOREBYPASS-S} \frac{\rho(pc) = \text{store } x, e \quad \text{speculate}(\mathcal{Y}, pc) \quad \omega > 0 \quad \omega' = \min(\omega - 1, \text{speculation-window}(\mathcal{Y}, pc)) \quad \mathcal{S}' = \mathcal{S} \triangleleft (\omega - 1, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \triangleleft (\omega', \varphi, \sigma, pc)}{\langle \varphi, \sigma, pc, \mathcal{Y}, \mathcal{S} \triangleleft (\omega, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \rangle \xrightarrow{\text{spec}_+} \langle \varphi, \sigma, pc + 1, \mathcal{Y}, \mathcal{S}' \rangle} \\
\\
\text{SPEC}_+ \text{TERMINATE} \frac{\rho(pc) = \perp \quad \omega > 0}{\langle \varphi, \sigma, pc, \mathcal{Y}, \mathcal{S} \triangleleft (\omega, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \rangle \xrightarrow{\text{spec}_+} \langle \varphi, \sigma, pc, \mathcal{Y}, \mathcal{S} \triangleleft (0, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \rangle} \\
\\
\text{SPEC}_+ \text{ROLLBACK} \frac{\mathcal{S} \triangleleft (0, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) = \mathcal{S}' \bowtie [(_, \widehat{\varphi}', \widehat{\sigma}', \widehat{pc}')] \bowtie \mathcal{S}'' \quad \widehat{pc} = \widehat{pc}' \quad (_, _, _, \widehat{pc}) \notin \mathcal{S}' \quad \langle \widehat{\varphi}', \widehat{\sigma}', \widehat{pc}' \rangle \longrightarrow \langle \varphi', \sigma', pc' \rangle}{\langle \varphi, \sigma, pc, \mathcal{Y}, \mathcal{S} \triangleleft (0, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \rangle \xrightarrow{\text{spec}_+} \langle \varphi', \sigma', pc', \mathcal{Y}, \mathcal{S}' \rangle}
\end{array}$$

A.3.1 Proof Idea

We want to show that the simpler `spec` semantics allows us to observe the same leaks as the more extensive `spec+` semantics. The motivation for using `spec` instead of `spec+` for our purpose is, that `spec` gives shorter traces and thus provides smaller counter-examples and additionally we expect that the resulting problem is “simpler” for the model checker and therefore more scalable.

Suppose that we have a program ρ with two conditional branches at the beginning, i.e. $\rho = [1 : \text{beqz } x, \ell_1; 2 : \text{beqz } y, \ell_2; \dots]$. We evaluate program ρ with and without nested speculation using both semantics to highlight the differences of both semantics in case of speculation during transient execution. Given $\varphi(x) = 0$, $\varphi(y) = 0$, `speculate`($\mathcal{Y}, 1$) and $\neg \text{taken}(\mathcal{Y}, 1)$.

Case 1: Assume that branch $\rho(2)$ isn’t speculated, i.e. $\neg \text{speculate}(\mathcal{Y}, 2)$.

$$\begin{aligned}
\tilde{\pi} &= \langle \varphi, \sigma, 1, \mathcal{Y}, \perp \rangle \xrightarrow{\text{SPECBRANCHPRED-NS}} \\
&\quad \langle \varphi, \sigma, 2, \mathcal{Y}, (\omega, \varphi, \sigma, 1) \rangle \xrightarrow{\text{SPECBRANCH-S}} \\
&\quad \langle \varphi, \sigma, \ell_2, \mathcal{Y}, (\omega - 1, \varphi, \sigma, 1) \rangle \xrightarrow{\text{spec}^*} \\
&\quad \langle \varphi', \sigma', pc', \mathcal{Y}, (0, \varphi, \sigma, 1) \rangle \xrightarrow{\text{SPECROLLBACK}} \\
&\quad \langle \varphi, \sigma, \ell_1, \mathcal{Y}, \perp \rangle
\end{aligned}$$

$$\begin{aligned}
\tilde{\pi}_+ &= \langle \varphi, \sigma, 1, \mathcal{Y}, [] \rangle \xrightarrow{\text{SPEC+BRANCHPRED-NS}} \\
&\quad \langle \varphi, \sigma, 2, \mathcal{Y}, [(\omega, \varphi, \sigma, 1)] \rangle \xrightarrow{\text{SPEC+BRANCH-S}} \\
&\quad \langle \varphi, \sigma, \ell_2, \mathcal{Y}, [(\omega - 1, \varphi, \sigma, 1)] \rangle \xrightarrow{\text{spec}_+^*} \\
&\quad \langle \varphi', \sigma', pc', \mathcal{Y}, [(0, \varphi, \sigma, 1)] \rangle \xrightarrow{\text{SPEC+ROLLBACK}} \\
&\quad \langle \varphi, \sigma, \ell_1, \mathcal{Y}, [] \rangle
\end{aligned}$$

Both semantics give the same program traces. Therefore, if $\tilde{\pi}_+$ contains a leak then $\tilde{\pi}$ contains the same leak.

Case 2: Assume that branch $\rho(2)$ is mis-predicted, i.e. $\text{speculate}(\mathcal{Y}, 2)$ and $\neg\text{taken}(\mathcal{Y}, 2)$.

$$\begin{aligned}
\tilde{\pi} &= \langle \varphi, \sigma, 1, \mathcal{Y}, \perp \rangle \xrightarrow{\text{SPECBRANCHPRED-NS}} \\
&\quad \langle \varphi, \sigma, 2, \mathcal{Y}, (\omega, \varphi, \sigma, 1) \rangle \xrightarrow{\text{SPECBRANCHPRED-S}} \\
&\quad \langle \varphi, \sigma, 3, \mathcal{Y}, (\omega - 1, \varphi, \sigma, 1) \rangle \xrightarrow{\text{spec}^*} \\
&\quad \langle \varphi'', \sigma'', pc'', \mathcal{Y}, (0, \varphi, \sigma, 1) \rangle \xrightarrow{\text{SPECROLLBACK}} \\
&\quad \langle \varphi, \sigma, \ell_1, \mathcal{Y}, \perp \rangle
\end{aligned}$$

$$\begin{aligned}
\tilde{\pi}_+ &= \langle \varphi, \sigma, 1, \mathcal{Y}, [] \rangle \xrightarrow{\text{SPEC+BRANCHPRED-NS}} \\
&\quad \langle \varphi, \sigma, 2, \mathcal{Y}, [(\omega, \varphi, \sigma, 1)] \rangle \xrightarrow{\text{SPEC+BRANCHPRED-S}} \\
&\quad \langle \varphi, \sigma, 3, \mathcal{Y}, [(\omega - 1, \varphi, \sigma, 1), (\omega', \varphi, \sigma, 2)] \rangle \xrightarrow{\text{spec}_+^*} \tag{A} \\
&\quad \langle \varphi'', \sigma'', pc'', \mathcal{Y}, [(\omega - 1, \varphi, \sigma, 1), (0, \varphi, \sigma, 2)] \rangle \xrightarrow{\text{SPEC+ROLLBACK}} \tag{B} \\
&\quad \langle \varphi, \sigma, \ell_2, \mathcal{Y}, [(\omega - 1, \varphi, \sigma, 1)] \rangle \xrightarrow{\text{spec}_+^*} \tag{C} \\
&\quad \langle \varphi', \sigma', pc', \mathcal{Y}, [(0, \varphi, \sigma, 1)] \rangle \xrightarrow{\text{SPEC+ROLLBACK}} \\
&\quad \langle \varphi, \sigma, \ell_1, \mathcal{Y}, [] \rangle
\end{aligned}$$

The trace given by the spec semantics is a subset of the trace given by the spec_+ semantics, as the re-evaluation of the nested mis-predicted branch $\rho(2)$ is missing from $\tilde{\pi}$. If $\tilde{\pi}_+$ contains a leak because of the nested mis-prediction (A),

then the same leak is also present in $\tilde{\pi}$. But what if $\tilde{\pi}_+$ contains a leak because of the re-evaluation of the mis-predicted branch $\rho(2)$? Then either resolving of the branch instruction during rollback (B) or the subsequently executed instructions (C) caused a leak. If we take a closer look at (C), we see that the instructions are executed as if no mis-prediction has occurred, because the register assignment and memory state from before the nested mis-prediction have been restored. Meaning that if (C) contains a leak after resolving the mis-prediction of branch $\rho(2)$, the same leak is also present if branch $\rho(2)$ has been correctly executed in the first place. By Appendix A.2 we have that the microarchitectural effects of the branch instruction in case of mis-prediction + rollback are the same as in case of correct execution. Therefore, if the leak has been caused by the rollback at (B), the same leak is also present if branch $\rho(2)$ is correctly predicted. By combining the last two observations we get that if $\tilde{\pi}_+$ contains a leak because of the re-evaluation of the mis-predicted branch $\rho(2)$, the same leak is also present when correctly executing branch $\rho(2)$ in the first place as done in **Case 1**. For $\text{taken}(\mathcal{Y}, 2)$ the reasoning is similar because our semantics also performs a rollback in case of correct prediction.

We have that the simpler spec semantics allows us to observe the same leaks as the more expensive spec_+ semantics, given two initial predictors \mathcal{Y}_1 and \mathcal{Y}_2 with $\text{speculate}(\mathcal{Y}_1, 2)$ respectively $\neg\text{speculate}(\mathcal{Y}_2, 2)$. Note that we don't constrain the predictor and therefore the solver can freely choose when and how to speculate and thus evaluate both cases.

A.3.2 Proof

By the definition of our semantics, the microarchitectural effects of each instruction solely depend on the current register assignment, current memory state, current program counter, next register assignment, next memory state and next program counter. Therefore we show that spec yields the same microarchitectural effects as spec_+ . From this follows that the spec semantics gives the same observations and consequently also the same leaks as the spec_+ semantics. Note that this proof requires that the effects are modeled as an unbounded set, such as a cache with unbounded size.

In the following we let $\text{eff}(\varphi, \sigma, pc, \varphi', \sigma', pc')$ denote a microarchitectural effect, where φ denotes the current register assignment, σ the current memory state, pc the current program counter, φ' the next register assignment, σ' the next memory state and pc' the next program counter. We let $\text{effects}(\pi)$ denote a set of such microarchitectural effects for the instructions on path π .

Induction Hypothesis: Given $\varphi, \sigma, pc, \varphi', \sigma', pc'$ and \mathcal{Y} both semantics spec as well as spec_+ yield the same effects during transient execution. More formally, we have that $E = E_+$ where $E = \text{effects}(\langle \varphi, \sigma, pc, \mathcal{Y}, \Delta \rangle \xrightarrow{\text{SPEC}}^* \langle \varphi', \sigma', pc', \mathcal{Y}, \Delta' \rangle)$ and $E_+ = \text{effects}(\langle \varphi, \sigma, pc, \mathcal{Y}, \mathcal{S} \rangle \xrightarrow{\text{SPEC}_+}^* \langle \varphi', \sigma', pc', \mathcal{Y}, \mathcal{S}' \rangle)$.

Base Case: Assume $\neg\text{speculate}(\mathcal{Y}, pc)$, meaning no nested speculation.

1. **Barrier instruction:**

$$\begin{aligned}\tilde{\pi}_+ &= \langle \varphi, \sigma, pc, \Upsilon, \mathcal{S} \rangle \xrightarrow{\text{SPEC}_+\text{BARRIER-S}} \langle \varphi, \sigma, pc + 1, \Upsilon, \mathcal{S}' \rangle \\ \tilde{\pi} &= \langle \varphi, \sigma, pc, \Upsilon, \Delta \rangle \xrightarrow{\text{SPECBARRIER-S}} \langle \varphi, \sigma, pc + 1, \Upsilon, \Delta' \rangle\end{aligned}$$

We obtain $\{\text{eff}(\varphi, \sigma, pc, \varphi, \sigma, pc + 1)\}$ for both, thus $\text{effects}(\tilde{\pi}) = \text{effects}(\tilde{\pi}_+)$.

2. **Observe instruction:**

$$\begin{aligned}\tilde{\pi}_+ &= \langle \varphi, \sigma, pc, \Upsilon, \mathcal{S} \rangle \xrightarrow{\text{SPEC}_+\text{OBSERVE-S}} \langle \varphi, \sigma, pc + 1, \Upsilon, \mathcal{S} \rangle \\ \tilde{\pi} &= \langle \varphi, \sigma, pc, \Upsilon, \Delta \rangle \xrightarrow{\text{SPECOBSERVE-S}} \langle \varphi, \sigma, pc + 1, \Upsilon, \Delta \rangle\end{aligned}$$

We obtain $\{\text{eff}(\varphi, \sigma, pc, \varphi, \sigma, pc + 1)\}$ for both, thus $\text{effects}(\tilde{\pi}) = \text{effects}(\tilde{\pi}_+)$.

3. **Lifted instructions:**

$$\begin{aligned}\tilde{\pi}_+ &= \langle \varphi, \sigma, pc, \Upsilon, \mathcal{S} \rangle \xrightarrow{\text{SPEC}_+\text{LIFT-S}} \langle \varphi', \sigma', pc', \Upsilon, \mathcal{S}' \rangle \\ &\quad \text{where } \varphi', \sigma' \text{ and } pc' \text{ is given by } \langle \varphi, \sigma, pc \rangle \longrightarrow \langle \varphi', \sigma', pc' \rangle \\ \tilde{\pi} &= \langle \varphi, \sigma, pc, \Upsilon, \Delta \rangle \xrightarrow{\text{SPECLIFT-S}} \langle \varphi'', \sigma'', pc'', \Upsilon, \Delta' \rangle \\ &\quad \text{where } \varphi'', \sigma'' \text{ and } pc'' \text{ is given by } \langle \varphi, \sigma, pc \rangle \longrightarrow \langle \varphi'', \sigma'', pc'' \rangle\end{aligned}$$

Given that the default semantics is deterministic we have that $\varphi' = \varphi'', \sigma' = \sigma''$ and $pc' = pc''$. We obtain $\{\text{eff}(\varphi, \sigma, pc, \varphi', \sigma', pc')\}$ for both, thus $\text{effects}(\tilde{\pi}) = \text{effects}(\tilde{\pi}_+)$.

4. **Branch instruction:**

$$\begin{aligned}\tilde{\pi}_+ &= \langle \varphi, \sigma, pc, \Upsilon, \mathcal{S} \rangle \xrightarrow{\text{SPEC}_+\text{BRANCH-S}} \langle \varphi, \sigma, pc', \Upsilon, \mathcal{S}' \rangle \\ &\quad \text{where } pc' \text{ is given by } \langle \varphi, \sigma, pc \rangle \longrightarrow \langle \varphi, \sigma, pc' \rangle \\ \tilde{\pi} &= \langle \varphi, \sigma, pc, \Upsilon, \Delta \rangle \xrightarrow{\text{SPECBRANCH-S}} \langle \varphi, \sigma, pc'', \Upsilon, \Delta' \rangle \\ &\quad \text{where } pc'' \text{ is given by } \langle \varphi, \sigma, pc \rangle \longrightarrow \langle \varphi, \sigma, pc'' \rangle\end{aligned}$$

Given that the default semantics is deterministic we have that $pc' = pc''$. We obtain $\{\text{eff}(\varphi, \sigma, pc, \varphi, \sigma, pc')\}$ for both, thus $\text{effects}(\tilde{\pi}) = \text{effects}(\tilde{\pi}_+)$.

5. **Store instruction:**

$$\begin{aligned}\tilde{\pi}_+ &= \langle \varphi, \sigma, pc, \Upsilon, \mathcal{S} \rangle \xrightarrow{\text{SPEC}_+\text{STORE-S}} \langle \varphi, \sigma', pc', \Upsilon, \mathcal{S}' \rangle \\ &\quad \text{where } \sigma' \text{ and } pc' \text{ is given by } \langle \varphi, \sigma, pc \rangle \longrightarrow \langle \varphi, \sigma', pc' \rangle \\ \tilde{\pi} &= \langle \varphi, \sigma, pc, \Upsilon, \Delta \rangle \xrightarrow{\text{SPECSTORE-S}} \langle \varphi, \sigma'', pc'', \Upsilon, \Delta' \rangle \\ &\quad \text{where } \sigma'' \text{ and } pc'' \text{ is given by } \langle \varphi, \sigma, pc \rangle \longrightarrow \langle \varphi, \sigma'', pc'' \rangle\end{aligned}$$

Given that the default semantics is deterministic we have that $\sigma' = \sigma''$ and $pc' = pc''$. We obtain $\{\text{eff}(\varphi, \sigma, pc, \varphi, \sigma', pc')\}$ for both, thus $\text{effects}(\tilde{\pi}) = \text{effects}(\tilde{\pi}_+)$.

As we have shown the equality of microarchitectural effects for each instruction, we have that for a sequence of non-speculated instructions $\text{effects}(\tilde{\pi}) = \text{effects}(\tilde{\pi}_+)$ holds.

Induction Step: Assume $\text{speculate}(\mathcal{I}, pc)$ for some transiently executed branch or store instruction.

1. **Branch instruction:** Assume $\varphi(x) = 0$, $\rho(pc) = \text{beqz } x, \ell$ and $\neg\text{taken}(\mathcal{I}, pc)$.

By spec_+ semantics we have a path $\tilde{\pi}_+$ which yields the effects $\text{effects}(\tilde{\pi}_+)$. We split the path into two cases, one for the nested speculation (A) and one for the resolved speculation (B). We show that for both cases the same effects are observable by the spec semantics.

$$\tilde{\pi}_+ = \left. \begin{array}{l} \langle \varphi, \sigma, pc, \mathcal{I}, \mathcal{S} \bowtie [(\omega, \hat{\varphi}, \hat{\sigma}, \hat{pc})] \rangle \xrightarrow{\text{SPEC}_+\text{BRANCHPRED-S}} \\ \langle \varphi, \sigma, pc + 1, \mathcal{I}, \mathcal{S} \bowtie [(\omega - 1, \hat{\varphi}, \hat{\sigma}, \hat{pc}), (\omega - 1, \varphi, \sigma, pc)] \rangle \xrightarrow{\text{SPEC}_+^*} \end{array} \right\} \text{A}$$

$$\left. \begin{array}{l} \langle \varphi', \sigma', pc', \mathcal{I}, \mathcal{S} \bowtie [(\omega - 1, \hat{\varphi}, \hat{\sigma}, \hat{pc}), (0, \varphi, \sigma, pc)] \rangle \xrightarrow{\text{SPEC}_+\text{ROLLBACK}} \\ \langle \varphi, \sigma, \ell, \mathcal{I}, \mathcal{S} \bowtie [(\omega - 1, \hat{\varphi}, \hat{\sigma}, \hat{pc})] \rangle \xrightarrow{\text{SPEC}_+^*} \\ \langle \varphi'', \sigma'', pc'', \mathcal{I}, \mathcal{S} \bowtie [(0, \hat{\varphi}, \hat{\sigma}, \hat{pc})] \rangle \end{array} \right\} \text{B}$$

$$\text{effects}(\tilde{\pi}_+) = \underbrace{\text{effects}(\tilde{\pi}_+^A)}_A \cup \underbrace{\text{effects}(\tilde{\pi}_+^B)}_B$$

Case A: Let $\tilde{\pi}_+^A$ be the sub-path covering the nested speculation. As the branch speculation doesn't have any immediate effects on the microarchitectural component (see μArch in Section 3.4), we have that $\text{effects}(\tilde{\pi}_+^A)$ corresponds to the effects of the following transiently executed instructions. Let $\tilde{\pi}^A$ be the path given by the spec semantics. By the deterministic behavior of both semantics we have that both yield the same final φ' , σ' and pc' .

$$\tilde{\pi}_+^A = \langle \varphi, \sigma, pc, \mathcal{I}, \mathcal{S} \bowtie [(\omega, \hat{\varphi}, \hat{\sigma}, \hat{pc})] \rangle \xrightarrow{\text{SPEC}_+\text{BRANCHPRED-S}}$$

$$\langle \varphi, \sigma, pc + 1, \mathcal{I}, \mathcal{S} \bowtie [(\omega - 1, \hat{\varphi}, \hat{\sigma}, \hat{pc}), (\omega - 1, \varphi, \sigma, pc)] \rangle \xrightarrow{\text{SPEC}_+^*}$$

$$\langle \varphi', \sigma', pc', \mathcal{I}, \mathcal{S} \bowtie [(\omega - 1, \hat{\varphi}, \hat{\sigma}, \hat{pc}), (0, \varphi, \sigma, pc)] \rangle$$

$$\text{effects}(\tilde{\pi}_+^A) = \underbrace{\text{effects}(\langle \varphi, \sigma, pc + 1, \mathcal{I}, \mathcal{S}' \rangle \xrightarrow{\text{SPEC}_+^*} \langle \varphi', \sigma', pc', \mathcal{I}, \mathcal{S}'' \rangle)}_{\text{IH}}$$

$$\tilde{\pi}^A = \langle \varphi, \sigma, pc, \mathcal{I}, (\omega, \hat{\varphi}, \hat{\sigma}, \hat{pc}) \rangle \xrightarrow{\text{SPEC}\text{BRANCHPRED-S}}$$

$$\langle \varphi, \sigma, pc + 1, \mathcal{I}, (\omega - 1, \hat{\varphi}, \hat{\sigma}, \hat{pc}) \rangle \xrightarrow{\text{SPEC}^*}$$

$$\langle \varphi', \sigma', pc', \mathcal{I}, (0, \hat{\varphi}, \hat{\sigma}, \hat{pc}) \rangle$$

$$\text{effects}(\tilde{\pi}^A) = \underbrace{\text{effects}(\langle \varphi, \sigma, pc + 1, \mathcal{I}, \Delta' \rangle \xrightarrow{\text{SPEC}^*} \langle \varphi', \sigma', pc', \mathcal{I}, \Delta'' \rangle)}_{\text{IH}}$$

By IH we obtain that $\text{effects}(\tilde{\pi}_+^A) = \text{effects}(\tilde{\pi}_+^A)$.

Case B: Let $\tilde{\pi}_+^B$ be the sub-path covering the resolved speculation. We have that $\text{effects}(\tilde{\pi}_+^B)$ consists of the effect of the resolved branch instruction as well as the effects of the subsequently executed instructions.

$$\begin{aligned} \tilde{\pi}_+^B &= \langle \varphi, \sigma, pc, \mathcal{Y}, \mathcal{S} \bowtie [(\omega, \hat{\varphi}, \hat{\sigma}, \hat{pc})] \rangle \xrightarrow{\text{SPEC}_+ \text{BRANCHPRED-S}} \\ &\quad \langle \varphi', \sigma', pc', \mathcal{Y}, \mathcal{S} \bowtie [(\omega - 1, \hat{\varphi}, \hat{\sigma}, \hat{pc}), (0, \varphi, \sigma, pc)] \rangle \xrightarrow{\text{SPEC}_+ \text{ROLLBACK}} \\ &\quad \langle \varphi, \sigma, \ell, \mathcal{Y}, \mathcal{S} \bowtie [(\omega - 1, \hat{\varphi}, \hat{\sigma}, \hat{pc})] \rangle \xrightarrow{\text{SPEC}_+^*} \\ &\quad \langle \varphi'', \sigma'', pc'', \mathcal{Y}, \mathcal{S} \bowtie [(0, \hat{\varphi}, \hat{\sigma}, \hat{pc})] \rangle \end{aligned}$$

By Appendix A.2 we have that the microarchitectural effects of the branch instruction in case of speculation + rollback are the same as in case of correct execution. Therefore we can replace the nested speculation in $\tilde{\pi}_+^B$ with a correct execution, while preserving its microarchitectural effects. Let \mathcal{Y}_r be a predictor which is equal to \mathcal{Y} expect that $\neg \text{speculate}(\mathcal{Y}_r, pc)$. Let $\tilde{\pi}_+^{B'}$ denote the transformed sub-path with predictor \mathcal{Y}_r . Note that this transformation is also valid in case of loops, because on rollback all nested speculative executions of the branch at pc are resolved. Let $\tilde{\pi}^B$ be the path given by the spec semantics. By the deterministic behavior of both semantics we have that both yield the same final φ'', σ'' and pc'' .

$$\begin{aligned} \tilde{\pi}_+^{B'} &= \langle \varphi, \sigma, pc, \mathcal{Y}_r, \mathcal{S} \bowtie [(\omega, \hat{\varphi}, \hat{\sigma}, \hat{pc})] \rangle \xrightarrow{\text{SPEC}_+ \text{BRANCH-S}} \\ &\quad \langle \varphi, \sigma, \ell, \mathcal{Y}_r, \mathcal{S} \bowtie [(\omega - 1, \hat{\varphi}, \hat{\sigma}, \hat{pc})] \rangle \xrightarrow{\text{SPEC}_+^*} \\ &\quad \langle \varphi'', \sigma'', pc'', \mathcal{Y}_r, \mathcal{S} \bowtie [(0, \hat{\varphi}, \hat{\sigma}, \hat{pc})] \rangle \\ \text{effects}(\tilde{\pi}_+^{B'}) &= \{\text{eff}(\varphi, \sigma, pc, \varphi, \sigma, \ell)\} \cup \\ &\quad \underbrace{\text{effects}(\langle \varphi, \sigma, \ell, \mathcal{Y}_r, \mathcal{S}' \rangle \xrightarrow{\text{SPEC}_+^*} \langle \varphi'', \sigma'', pc'', \mathcal{Y}_r, \mathcal{S}'' \rangle)}_{\text{IH}} \\ \tilde{\pi}^B &= \langle \varphi, \sigma, pc, \mathcal{Y}_r, (\omega, \hat{\varphi}, \hat{\sigma}, \hat{pc}) \rangle \xrightarrow{\text{SPECBRANCH-S}} \\ &\quad \langle \varphi, \sigma, \ell, \mathcal{Y}_r, (\omega - 1, \hat{\varphi}, \hat{\sigma}, \hat{pc}) \rangle \xrightarrow{\text{SPEC}^*} \\ &\quad \langle \varphi'', \sigma'', pc'', \mathcal{Y}_r, (0, \hat{\varphi}, \hat{\sigma}, \hat{pc}) \rangle \\ \text{effects}(\tilde{\pi}^B) &= \{\text{eff}(\varphi, \sigma, pc, \varphi, \sigma, \ell)\} \cup \\ &\quad \underbrace{\text{effects}(\langle \varphi, \sigma, \ell, \mathcal{Y}_r, \Delta' \rangle \xrightarrow{\text{SPEC}^*} \langle \varphi'', \sigma'', pc'', \mathcal{Y}_r, \Delta'' \rangle)}_{\text{IH}} \end{aligned}$$

By IH and equality of $\text{eff}(\varphi, \sigma, pc, \varphi, \sigma, \ell)$ we obtain that $\text{effects}(\tilde{\pi}^B) = \text{effects}(\tilde{\pi}_+^{B'})$. As the transformation of $\tilde{\pi}_+^B$ was effect preserving we furthermore obtain that $\text{effects}(\tilde{\pi}^B) = \text{effects}(\tilde{\pi}_+^B)$.

We have $\text{effects}(\tilde{\pi}^A) = \text{effects}(\tilde{\pi}_+^A)$ and $\text{effects}(\tilde{\pi}^B) = \text{effects}(\tilde{\pi}_+^B)$. Consequently it holds that $\text{effects}(\tilde{\pi}_+) = \text{effects}(\tilde{\pi}_+^A) \cup \text{effects}(\tilde{\pi}_+^B)$. We can re-formulate it as $\text{effects}(\tilde{\pi}_+) = \text{effects}(\tilde{\pi}_{\text{speculate}(pc)}) \cup \text{effects}(\tilde{\pi}_{\text{-speculate}(pc)})$, meaning that for `spec` semantics the transient execution started at \widehat{pc} needs to be evaluated twice, with and without speculation for the transiently executed branch instruction at pc , to observe the same effects. For $\varphi(x) \neq 0$ respectively `taken`(\mathcal{T}, pc) the proof is similar.

2. Store instruction: Assume $\rho(pc) = \text{store } x, e$.

By `spec+` semantics we have a path $\tilde{\pi}_+$ which yields the effects $\text{effects}(\tilde{\pi}_+)$. We split the path into two cases, one for the nested speculation (A) and one for the resolved speculation (B). We show that for both cases the same effects are observable by the `spec` semantics.

$$\tilde{\pi}_+ = \left. \begin{array}{l} \langle \varphi, \sigma, pc, \mathcal{T}, \mathcal{S} \bowtie [(\omega, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc})] \rangle \xrightarrow{\text{SPEC}_+ \text{STOREBYPASS-S}} \\ \langle \varphi, \sigma, pc + 1, \mathcal{T}, \mathcal{S} \bowtie [(\omega - 1, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}), (\omega - 1, \varphi, \sigma, pc)] \rangle \xrightarrow{\text{SPEC}_+^*} \end{array} \right\} \text{A}$$

$$\left. \begin{array}{l} \langle \varphi', \sigma', pc', \mathcal{T}, \mathcal{S} \bowtie [(\omega - 1, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}), (0, \varphi, \sigma, pc)] \rangle \xrightarrow{\text{SPEC}_+ \text{ROLLBACK}} \\ \langle \varphi, \sigma^r, pc + 1, \mathcal{T}, \mathcal{S} \bowtie [(\omega - 1, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc})] \rangle \xrightarrow{\text{SPEC}_+^*} \\ \langle \varphi'', \sigma'', pc'', \mathcal{T}, \mathcal{S} \bowtie [(0, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc})] \rangle \end{array} \right\} \text{B}$$

where $\sigma^r = \sigma[[e]\varphi \mapsto \varphi(x)]$

$$\text{effects}(\tilde{\pi}_+) = \underbrace{\text{effects}(\tilde{\pi}_+^A)}_A \cup \underbrace{\text{effects}(\tilde{\pi}_+^B)}_B$$

Case A: Let $\tilde{\pi}_+^A$ be the sub-path covering the nested speculation. As the bypassed store doesn't have any immediate effects on the microarchitectural component (see μArch in Section 3.4), we have that $\text{effects}(\tilde{\pi}_+^A)$ corresponds to the effects of the following transiently executed instructions. Let $\tilde{\pi}^A$ be the path given by the `spec` semantics. By the deterministic behavior

of both semantics we have that both yield the same final φ' , σ' and pc' .

$$\begin{aligned}
\tilde{\pi}_+^A &= \langle \varphi, \sigma, pc, \mathcal{Y}, \mathcal{S} \bowtie [(\omega, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc})] \rangle \xrightarrow{\text{SPEC}_+ \text{STOREBYPASS-S}} \\
&\quad \langle \varphi, \sigma, pc + 1, \mathcal{Y}, \mathcal{S} \bowtie [(\omega - 1, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}), (\omega - 1, \varphi, \sigma, pc)] \rangle \xrightarrow{\text{SPEC}_+^*} \\
&\quad \langle \varphi', \sigma', pc', \mathcal{Y}, \mathcal{S} \bowtie [(\omega - 1, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}), (0, \varphi, \sigma, pc)] \rangle \\
\text{effects}(\tilde{\pi}_+^A) &= \underbrace{\text{effects}(\langle \varphi, \sigma, pc + 1, \mathcal{Y}, \mathcal{S}' \rangle \xrightarrow{\text{SPEC}_+^*} \langle \varphi', \sigma', pc', \mathcal{Y}, \mathcal{S}'' \rangle)}_{\text{IH}} \\
\tilde{\pi}^A &= \langle \varphi, \sigma, pc, \mathcal{Y}, (\omega, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \rangle \xrightarrow{\text{SPECSTOREBYPASS-S}} \\
&\quad \langle \varphi, \sigma, pc + 1, \mathcal{Y}, (\omega - 1, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \rangle \xrightarrow{\text{SPEC}^*} \\
&\quad \langle \varphi', \sigma', pc', \mathcal{Y}, (0, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \rangle \\
\text{effects}(\tilde{\pi}^A) &= \underbrace{\text{effects}(\langle \varphi, \sigma, pc + 1, \mathcal{Y}, \Delta' \rangle \xrightarrow{\text{SPEC}^*} \langle \varphi', \sigma', pc', \mathcal{Y}, \Delta'' \rangle)}_{\text{IH}}
\end{aligned}$$

By IH we obtain that $\text{effects}(\tilde{\pi}^A) = \text{effects}(\tilde{\pi}_+^A)$.

Case B: Let $\tilde{\pi}_+^B$ be the sub-path covering the resolved speculation. We have that $\text{effects}(\tilde{\pi}_+^B)$ consists of the effect of the resolved store instruction as well as the effects of the subsequently executed instructions.

$$\begin{aligned}
\tilde{\pi}_+^B &= \langle \varphi, \sigma, pc, \mathcal{Y}, \mathcal{S} \bowtie [(\omega, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc})] \rangle \xrightarrow{\text{SPEC}_+ \text{STOREBYPASS-S}} \\
&\quad \langle \varphi', \sigma', pc', \mathcal{Y}, \mathcal{S} \bowtie [(\omega - 1, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}), (0, \varphi, \sigma, pc)] \rangle \xrightarrow{\text{SPEC}_+ \text{ROLLBACK}} \\
&\quad \langle \varphi, \sigma^r, pc + 1, \mathcal{Y}, \mathcal{S} \bowtie [(\omega - 1, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc})] \rangle \xrightarrow{\text{SPEC}_+^*} \\
&\quad \langle \varphi'', \sigma'', pc'', \mathcal{Y}, \mathcal{S} \bowtie [(0, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc})] \rangle \\
&\quad \text{where } \sigma^r = \sigma[[e]]\varphi \mapsto \varphi(x)
\end{aligned}$$

By Appendix A.2 we have that the microarchitectural effects of the store instruction in case of speculation + rollback are the same as in case of correct execution. Therefore we can replace the nested speculation in $\tilde{\pi}_+^B$ with a correct execution, while preserving its microarchitectural effects. Let \mathcal{Y}_r be a predictor which is equal to \mathcal{Y} expect that $\neg \text{speculate}(\mathcal{Y}_r, pc)$. Let $\tilde{\pi}_+^{B'}$ denote the transformed sub-path with predictor \mathcal{Y}_r . Note that this transformation is also valid in case of loops, because on rollback all nested speculative executions of the branch at pc are resolved. Let $\tilde{\pi}^B$ be the path given by the spec semantics. By the deterministic behavior of

both semantics we have that both yield the same final φ'' , σ'' and pc'' .

$$\begin{aligned}
\tilde{\pi}_+^{B'} &= \langle \varphi, \sigma, pc, \Upsilon_r, \mathcal{S} \bowtie [(\omega, \hat{\varphi}, \hat{\sigma}, \hat{pc})] \rangle \xrightarrow{\text{SPEC}_+ \text{STORE-S}} \\
&\quad \langle \varphi, \sigma^r, pc+1, \Upsilon_r, \mathcal{S} \bowtie [(\omega-1, \hat{\varphi}, \hat{\sigma}, \hat{pc})] \rangle \xrightarrow{\text{SPEC}_+^*} \\
&\quad \langle \varphi'', \sigma'', pc'', \Upsilon_r, \mathcal{S} \bowtie [(0, \hat{\varphi}, \hat{\sigma}, \hat{pc})] \rangle \\
&\quad \text{where } \sigma^r = \sigma[[e]]\varphi \mapsto \varphi(x) \\
\text{effects}(\tilde{\pi}_+^{B'}) &= \{\text{eff}(\varphi, \sigma, pc, \varphi, \sigma^r, pc+1)\} \cup \\
&\quad \underbrace{\text{effects}(\langle \varphi, \sigma^r, pc+1, \Upsilon_r, \mathcal{S}' \rangle \xrightarrow{\text{SPEC}_+^*} \langle \varphi'', \sigma'', pc'', \Upsilon_r, \mathcal{S}'' \rangle)}_{\text{IH}} \\
\tilde{\pi}^B &= \langle \varphi, \sigma, pc, \Upsilon_r, (\omega, \hat{\varphi}, \hat{\sigma}, \hat{pc}) \rangle \xrightarrow{\text{SPECSTORE-S}} \\
&\quad \langle \varphi, \sigma^r, pc+1, \Upsilon_r, (\omega-1, \hat{\varphi}, \hat{\sigma}, \hat{pc}) \rangle \xrightarrow{\text{SPEC}^*} \\
&\quad \langle \varphi'', \sigma'', pc'', \Upsilon_r, (0, \hat{\varphi}, \hat{\sigma}, \hat{pc}) \rangle \\
\text{effects}(\tilde{\pi}^B) &= \{\text{eff}(\varphi, \sigma, pc, \varphi, \sigma^r, pc+1)\} \cup \\
&\quad \underbrace{\text{effects}(\langle \varphi, \sigma^r, pc+1, \Upsilon_r, \Delta' \rangle \xrightarrow{\text{SPEC}^*} \langle \varphi'', \sigma'', pc'', \Upsilon_r, \Delta'' \rangle)}_{\text{IH}}
\end{aligned}$$

By IH and equality of $\text{eff}(\varphi, \sigma, pc, \varphi, \sigma^r, pc+1)$ we obtain that $\text{effects}(\tilde{\pi}^B) = \text{effects}(\tilde{\pi}_+^{B'})$. As the transformation of $\tilde{\pi}_+^{B'}$ was effect preserving we furthermore obtain that $\text{effects}(\tilde{\pi}^B) = \text{effects}(\tilde{\pi}_+^B)$.

We have $\text{effects}(\tilde{\pi}^A) = \text{effects}(\tilde{\pi}_+^A)$ and $\text{effects}(\tilde{\pi}^B) = \text{effects}(\tilde{\pi}_+^B)$. Consequently it holds that $\text{effects}(\tilde{\pi}_+) = \text{effects}(\tilde{\pi}_+^A) \cup \text{effects}(\tilde{\pi}_+^B)$. We can re-formulate it as $\text{effects}(\tilde{\pi}_+) = \text{effects}(\tilde{\pi}_{\text{speculate}(pc)}) \cup \text{effects}(\tilde{\pi}_{\neg\text{speculate}(pc)})$, meaning that for spec semantics the transient execution started at \hat{pc} needs to be evaluated twice, with and without speculation for the transiently executed store instruction at pc , to observe the same effects.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Miscellaneous

B.1 Predictor Comparison

In this section we compare our taken/not-taken predictor to the mis-predict predictor as used in Spectector [21]. To avoid any confusions with our definitions from Section 3.3, we denote the mis-predict predictor as mis-speculate predictor. First we give a short definition of the branching behavior of both predictors, then we show that depending on the adversary's observational capabilities, the mis-speculate predictor may not be sufficient to catch all transient execution vulnerabilities.

B.1.1 Taken/Not-Taken Predictor

The predicate $\text{speculate}(\mathcal{Y}, pc)$ denotes if speculative execution should be started at program location pc . In real attacks the adversary would for example flush specific values from the cache to trigger the speculative execution. The predicate $\text{taken}(\mathcal{Y}, pc)$ denotes if a branch, when executed speculatively, should be taken or not. This models the behavior of the branch predictor, more specifically the behavior of the PHT. In real attacks the adversary would for example train the branch predictor accordingly to either take or not take the branch.

$$\begin{array}{c}
 \text{SPECBRANCH-NS} \frac{\rho(pc) = \text{beqz } x, \ell \quad \neg \text{speculate}(\mathcal{Y}, pc) \quad \langle \varphi, \sigma, pc \rangle \longrightarrow \langle \varphi', \sigma', pc' \rangle}{\langle \varphi, \sigma, pc, \mathcal{Y}, \perp \rangle \xrightarrow{\text{spec}} \langle \varphi', \sigma', pc', \mathcal{Y}, \perp \rangle} \\
 \\
 \text{SPECBRANCHPRED-NS} \frac{\begin{array}{c} \rho(pc) = \text{beqz } x, \ell \\ \text{speculate}(\mathcal{Y}, pc) \quad pc' = \text{ite}(\text{taken}(\mathcal{Y}, pc), \ell, pc + 1) \\ \omega' = \text{speculation-window}(\mathcal{Y}, pc) \end{array}}{\langle \varphi, \sigma, pc, \mathcal{Y}, \perp \rangle \xrightarrow{\text{spec}} \langle \varphi, \sigma, pc', \mathcal{Y}, (\omega', \varphi, \sigma, pc) \rangle}
 \end{array}$$

$$\text{SPECBRANCH-S} \frac{\rho(pc) = \text{beqz } x, \ell \quad \neg \text{speculate}(\mathcal{Y}, pc) \quad \omega > 0 \quad \langle \varphi, \sigma, pc \rangle \longrightarrow \langle \varphi', \sigma', pc' \rangle}{\langle \varphi, \sigma, pc, \mathcal{Y}, (\omega, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \rangle \xrightarrow{\text{spec}} \langle \varphi', \sigma', pc', \mathcal{Y}, (\omega - 1, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \rangle}$$

$$\text{SPECBRANCHPRED-S} \frac{\rho(pc) = \text{beqz } x, \ell \quad \text{speculate}(\mathcal{Y}, pc) \quad \omega > 0 \quad pc' = \text{ite}(\text{taken}(\mathcal{Y}, pc), \ell, pc + 1)}{\langle \varphi, \sigma, pc, \mathcal{Y}, (\omega, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \rangle \xrightarrow{\text{spec}} \langle \varphi, \sigma, pc', \mathcal{Y}, (\omega - 1, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \rangle}$$

Note that SPECBRANCHPRED-NS as defined above may also transiently execute correctly predicted paths. Meaning that a path is first transiently executed until the speculation limit is reached and then after rollback the exact same path is re-executed. The reason for that is, that we only model “rollback” but not “commit”. While this doesn’t affect the correctness of the semantics, it may introduce unnecessary paths which need to be examined by the solver.

B.1.2 Mis-Speculate Predictor

The predicate $\text{mis-speculate}(\mathcal{Y}, pc)$ denotes if the instruction at program location pc should be mis-speculated. By $\text{mis-speculate}(\mathcal{Y}, pc)$ the adversary can control for which instruction mis-speculation should be enforced, in real attacks the adversary would for example flush the cache and train the branch predictor accordingly.

The branch instruction may mis-predict the branch decision, meaning that the opposite decision compared to the non-speculative semantics is taken if the predicate $\text{mis-speculate}(\mathcal{Y}, pc)$ holds. Therefore, in contrast to the taken/not-taken predictor, the path taken during mis-speculation depends on the actual condition value instead of adversary controllable input.

$$\text{SPECBRANCH-NS} \frac{\rho(pc) = \text{beqz } x, \ell \quad \neg \text{mis-speculate}(\mathcal{Y}, pc) \quad \langle \varphi, \sigma, pc \rangle \longrightarrow \langle \varphi', \sigma', pc' \rangle}{\langle \varphi, \sigma, pc, \mathcal{Y}, \perp \rangle \xrightarrow{\text{spec}} \langle \varphi', \sigma', pc', \mathcal{Y}, \perp \rangle}$$

$$\text{SPECBRANCHMIS-NS} \frac{\rho(pc) = \text{beqz } x, \ell \quad \text{mis-speculate}(\mathcal{Y}, pc) \quad pc' = \text{ite}(\varphi(x) = 0, pc + 1, \ell) \quad \omega' = \text{speculation-window}(\mathcal{Y}, pc)}{\langle \varphi, \sigma, pc, \mathcal{Y}, \perp \rangle \xrightarrow{\text{spec}} \langle \varphi, \sigma, pc', \mathcal{Y}, (\omega', \varphi, \sigma, pc) \rangle}$$

$$\text{SPECBRANCH-S} \frac{\rho(pc) = \text{beqz } x, \ell \quad \neg \text{mis-speculate}(\mathcal{Y}, pc) \quad \omega > 0 \quad \langle \varphi, \sigma, pc \rangle \longrightarrow \langle \varphi', \sigma', pc' \rangle}{\langle \varphi, \sigma, pc, \mathcal{Y}, (\omega, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \rangle \xrightarrow{\text{spec}} \langle \varphi', \sigma', pc', \mathcal{Y}, (\omega - 1, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \rangle}$$

$$\text{SPECBRANCHMIS-S} \frac{\rho(pc) = \text{beqz } x, \ell \quad \text{mis-speculate}(\mathcal{Y}, pc) \quad \omega > 0 \quad pc' = \text{ite}(\varphi(x) = 0, pc + 1, \ell)}{\langle \varphi, \sigma, pc, \mathcal{Y}, (\omega, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \rangle \xrightarrow{\text{spec}} \langle \varphi, \sigma, pc', \mathcal{Y}, (\omega - 1, \widehat{\varphi}, \widehat{\sigma}, \widehat{pc}) \rangle}$$

B.1.3 Comparison

$\text{beqz } x, \ell$	$\text{beqz } x^d, \ell$	Mis-Speculate	Taken/Not-Taken
$\varphi(x) = 0$	$\varphi(x^d) \neq 0$	$\neg \text{mis-speculate}(\mathcal{T}, pc)$ 	$\neg \text{speculate}(\mathcal{T}, pc)$
	$\varphi(x^d) = 0$	$\text{mis-speculate}(\mathcal{T}, pc)$ 	$\text{speculate}(\mathcal{T}, pc) \wedge \text{taken}(\mathcal{T}, pc)$ $\text{speculate}(\mathcal{T}, pc) \wedge \neg \text{taken}(\mathcal{T}, pc)$

Table B.1: Path Combinations of Mis-Speculate and Taken/Not-Taken Predictors for Self-Composed Branch Instruction (correct paths are green, transient paths are red).

In Table B.1 the path combinations of both predictors for a self-composed branch instruction $\text{beqz } x, \ell$ are shown. The cases $\varphi(x) \neq 0 \wedge \varphi(x^d) = 0$ and $\varphi(x) \neq 0 \wedge \varphi(x^d) \neq 0$ are exactly contrary to the listed cases and therefore intentionally left out.

Without speculative execution, meaning neither $\text{mis-speculate}(\mathcal{T}, pc)$ nor $\text{speculate}(\mathcal{T}, pc)$ holds, both predictors give the same path combinations in both cases. Additionally, when the branch conditions are equal, meaning $\varphi(x) = 0$ and $\varphi(x^d) = 0$, both predictors give the same path combinations even in the presence of speculative execution.

The only case where the predictors give different path combinations is, when the

branch is mis-predicted respectively speculatively executed and the branch conditions are unequal, meaning $\varphi(x) = 0$ and $\varphi(x^d) \neq 0$, which can only happen if the branch condition depends on high-security input.

In a nutshell, the predictors give dissimilar path combinations when the control-flow depends on high-security input with distinct condition values. This is problematic for the mis-speculate predictor, as some transient execution leaks may be missed under specific conditions.

- Suppose that the adversary cannot directly observe the program counter, meaning that the branch condition can only be leaked indirectly.

- Assume that $\begin{matrix} pc \\ \swarrow \searrow \\ pc+1 \quad \ell \end{matrix} \approx_{\text{Obs}} \begin{matrix} pc \\ \swarrow \searrow \\ pc+1 \quad \ell \end{matrix}$ and $\begin{matrix} pc \\ \swarrow \searrow \\ pc+1 \quad \ell \end{matrix} \approx_{\text{Obs}} \begin{matrix} pc \\ \swarrow \searrow \\ pc+1 \quad \ell \end{matrix}$ holds, meaning that the instructions on both branches yield equal observations when both are executed correctly or both are executed transiently.

- This assumption doesn't prevent that $\begin{matrix} pc \\ \swarrow \searrow \\ pc+1 \quad \ell \end{matrix} \not\approx_{\text{Obs}} \begin{matrix} pc \\ \swarrow \searrow \\ pc+1 \quad \ell \end{matrix}$ respectively $\begin{matrix} pc \\ \swarrow \searrow \\ pc+1 \quad \ell \end{matrix} \not\approx_{\text{Obs}} \begin{matrix} pc \\ \swarrow \searrow \\ pc+1 \quad \ell \end{matrix}$ holds, meaning that the instructions on both branches yield differing observations when one is executed correctly while the other is executed transiently.

In such a scenario only the taken/not-taken predictor is able to detect the transient execution leaks. The mis-speculate predictor fails to detect the transient execution leaks because of our assumption that both paths yield the same observations during transient execution.

B.1.4 Example

```

1 int x = nondet() ? 1 : -1;
2 int zero = 0;
3
4 void victim() {
5     if (...) {
6         // transient execution
7         if (x >= zero) {
8             v = x;
9         } else {
10            v = -x;
11        }
12        cache_encode(v);
13    }
14 }

```

Listing B.1: High-Security Conditional

```

1 int xd = nondet() ? 1 : -1;
2 int zerod = 0;
3
4 void victimd() {
5     if (...) {
6         // transient execution
7         if (xd >= zerod) {
8             vd = xd;
9         } else {
10            vd = -xd;
11        }
12        cache_encode(vd);
13    }
14 }

```

Listing B.2: Self-Composition of B.1

Listings B.1 and B.2 show a small example to demonstrate the difference of both predictors as explained above. We have a high-security variable x respectively x^d . W.l.o.g. assume that $x = +1$ and $x^d = -1$. We show for both predictors how the variables v and v^d as well as the program counter evolves from line 7 to 12.

1. Mis-Speculate Predictor:

Case A: Assume that $\neg \text{mis-speculate}(\mathcal{T}, 7)$.

$$\begin{aligned}
 (v : \perp, pc : 7) &\rightarrow (v : \perp, pc : 8) \rightarrow (v : +1, pc : 12) \\
 (v^d : \perp, pc^d : 7) &\rightarrow (v^d : \perp, \underbrace{pc^d : 10}_{pc \neq pc^d}) \rightarrow (v^d : +1, pc^d : 12)
 \end{aligned}$$

Case B: Assume that $\text{mis-speculate}(\mathcal{T}, 7)$.

$$\begin{aligned}
 (v : \perp, pc : 7) &\rightarrow (v : \perp, pc : 10) \rightarrow (v : -1, pc : 12) \\
 (v^d : \perp, pc^d : 7) &\rightarrow (v^d : \perp, \underbrace{pc^d : 8}_{pc \neq pc^d}) \rightarrow (v^d : -1, pc^d : 12)
 \end{aligned}$$

In both cases information about x is leaked via pc . An adversary who can observe the control-flow, for example through the branch-predictor, is able to learn sensitive information about x . We get that with the mis-speculate predictor the adversary needs to be able to observe the control-flow to obtain the leak. For an adversary who can only observe memory accesses, the program shown in Listing B.1 will be marked as secure!

2. Taken/Not-Taken Predictor:

Case A: Assume that $\neg\text{speculate}(\mathcal{I}, 7)$.

$$\begin{aligned} (v : \perp, pc : 7) \rightarrow (v : \perp, pc : 8) \rightarrow (v : +1, pc : 12) \\ (v^d : \perp, pc^d : 7) \rightarrow (v^d : \perp, \underbrace{pc^d : 10}_{pc \neq pc^d}) \rightarrow (v^d : +1, pc^d : 12) \end{aligned}$$

In this case information about x is leaked via pc . An adversary who can observe the control-flow, for example through the branch-predictor, is able to learn sensitive information about x .

Case B: Assume that $\text{speculate}(\mathcal{I}, 7)$ and $\text{taken}(\mathcal{I}, 7)$.

$$\begin{aligned} (v : \perp, pc : 7) \rightarrow (v : \perp, pc : 10) \rightarrow (v : -1, pc : 12) \\ (v^d : \perp, pc^d : 7) \rightarrow (v^d : \perp, pc^d : 10) \rightarrow (\underbrace{v^d : +1}_{v \neq v^d}, pc^d : 12) \end{aligned}$$

In this case information about x is leaked via v . An adversary who can observe the value of v , for example by leaking it through a cache-based covert-channel, is able to learn sensitive information about x .

Case C: Assume that $\text{speculate}(\mathcal{I}, 7)$ and $\neg\text{taken}(\mathcal{I}, 7)$.

$$\begin{aligned} (v : \perp, pc : 7) \rightarrow (v : \perp, pc : 8) \rightarrow (v : +1, pc : 12) \\ (v^d : \perp, pc^d : 7) \rightarrow (v^d : \perp, pc^d : 8) \rightarrow (\underbrace{v^d : -1}_{v \neq v^d}, pc^d : 12) \end{aligned}$$

In this case information about x is leaked via v . An adversary who can observe the value of v , for example by leaking it through a cache-based covert-channel, is able to learn sensitive information about x .

We get that the leak is visible to an adversary who can observe the control-flow and/or memory accesses. In contrast to the mis-speculate predictor shown in 1., the taken/not-taken predictor marks the program shown in Listing B.1 as insecure, even when the adversary can e.g. only observe the cache.

B.2 SpecBMC Environment Reference

In this section we provide the reference of SpecBMC's environment file:

```
# HIR/LIR optimization level: none, basic, full [default: full]
# - none: no optimizations
# - basic: copy propagation
# - full: constant folding & propagation, expression
#         simplification and copy propagation
optimization: full
```

```

# SMT solver: z3, cvc4, yices2 [default: yices2]
solver: yices2

# Analysis
analysis:
  # Search for Spectre-PHT? false, true [default: true]
  spectre_pht: true
  # Search for Spectre-STL? false, true [default: false]
  spectre_stl: false
  # Type of leak check: [default: only_transient_leaks]
  #   - only_transient_leaks: Only find leaks which are there
  #                           because of transient execution
  #   - only_normal_leaks: Find normal leaks (no transient
  #                           execution)
  #   - all_leaks: Search for both types of leaks
  check: only_transient_leaks
  # Branch prediction strategy: [default: choose_path]
  #   - choose_path: predict taken/not-taken
  #   - invert_condition: mis-predict (take the opposite)
  predictor_strategy: choose_path
  # The default number of loop iterations to unwind: n >= 0
  unwind: 0
  # The number of loop iterations to unwind for specific loops
  # (key is loop id, value is unwinding bound >= 0).
  # If no specific loop bound is given, the default unwinding bound
  # is used instead.
  unwind_loop:
    ...
  # Use unwinding assumptions or assertions [default: assumption]
  unwinding_guard: assumption
  # Recursion limit for recursive function-inlining: [default: 0]
  recursion_limit: 0
  # Start with empty (flushed) cache? false, true [default: false]
  # Note: Only available when using CVC4 or Z3.
  start_with_empty_cache: false
  # Type of observation: [default: parallel]
  #   - sequential: Observe only at the end of the program.
  #   - parallel: Observe each instruction and control-flow join.
  #   - trace: Observe trace of side effects.
  observe: parallel
  # Type of analysis model: components, pc [default: components]
  #   - components: Observe microarchitectual components like
  #                 cache, branch-target buffer, ...
  #   - pc: Observe program counter and memory locations
  model: components
  # The program entry point: string [default: entry point from ELF]
  program_entry: "main"
  # List of function names which should not be inlined
  inline_ignore: []

```

```
# Architecture
architecture:
  # Is cache available to attacker? [default: true]
  cache: true
  # Is branch target buffer available to attacker? [default: true]
  btb: true
  # Is pattern history table available to attacker? [default: true]
  pht: true
  # The length of the speculation window: n >= 0 [default: 100]
  speculation_window: 100

# Security policy
policy:
  registers:
    # Default security policy of all registers: low, high
    default: low
    # List of high-security registers
    # (only makes sense when default is low)
    high: []
    # List of low-security registers
    # (only makes sense when default is high)
    low: []
  memory: # Sections with start and end address (end is exclusive)
    # Default security policy of all memory locations: low, high
    default: high
    # List of high-security memory locations
    # (only makes sense when default is low)
    high: []
    # List of low-security memory locations
    # (only makes sense when default is high)
    low: []

# Initial Setup
setup:
  # Prepare stack (0xffff_0000_0000 < rsp <= rbp) and return addr
  init_stack: false
  # Initial register content (key is name, value is content)
  registers:
    ...
  # Initial flag register content (key is name, value is state)
  flags:
    ...
  # Initial memory content (key is address, value is content)
  memory:
    ...

# Debug mode: false, true [default: false]
debug: false
```

List of Figures

2.1	From Binary to μ ASM	9
3.1	Out-of-order Execution and Retirement	13
3.2	Simplified Transient Execution Attack Classification Tree [7]	14
3.3	Cause of Transient Execution (adopted from [18])	15
3.4	Four Phases of a Transient Execution Attack	16
3.5	Control-Flow Graph and Transient Attack Path of Victim Function	18
3.6	Simplified Memory Instruction Sequence of Victim Function	21
4.1	From Memory Address to Cache Tag, Index and Block Offset	31
4.2	Cache State Depending on the Least-Significant Bit of y	34
4.3	Cache Example Traces for $\text{LSB}(y) = 1$ and $\text{LSB}(y) = 0$	35
4.4	AVX Unit State Depending on the Least-Significant Bit of y	37
4.5	AVX Unit Example Traces for $\text{LSB}(y) = 1$ and $\text{LSB}(y) = 0$	38
4.6	BTB State Depending on the Least-Significant Bit of y	42
4.7	PHT State Depending on the Least-Significant Bit of y	42
4.8	BTB Example Traces for $\text{LSB}(y) = 1$ and $\text{LSB}(y) = 0$	43
5.1	Architecture of SpecBMC	45
5.2	High-level Intermediate Representation of Kocher01 Example	49
5.3	Function Inlining Example	53
5.4	Loop Unrolling Example	53
5.5	Transient Execution Transformation Example	54
5.6	Observation Models applied to Kocher01 Example	56
5.7	SSA Transformation Example	58
5.8	Transformed High-level Intermediate Representation of Kocher01 Example	61
5.9	Mid-level Intermediate Representation of Kocher01 Example	64
5.10	Counterexample for Kocher01 Example	73
5.11	Program Counter Model applied to Kocher01 Example	74
6.1	Assembly Code of Spectre-PHT Victim Function	79
6.2	Assembly Code of Kocher Example 10 (O2, SLH)	87
6.3	Assembly Code of Kocher Example 15 (O0, SLH)	89
6.4	Execution Time of SpecBMC using different Settings (Examples 1-5)	90
6.5	Execution Time of SpecBMC using different Settings (Examples 6-10)	91

6.6	Execution Time of SpecBMC using different Settings (Examples 11-15)	92
6.7	Execution Time of SpecBMC and Spectector (Examples 1-5)	93
6.8	Execution Time of SpecBMC and Spectector (Examples 6-10)	94
6.9	Execution Time of SpecBMC and Spectector (Examples 11-15)	95
6.10	Simplified Assembly Code of <code>ptrace_set_debugreg()</code>	102
7.1	Spectre-PHT Information Leak in Spectector	104

List of Tables

6.1	Hard- and Software Configuration	75
6.2	Evaluation Results for Spectre-STL + Mitigation	78
6.3	Evaluation Results for Spectre-PHT + Mitigation	82
6.4	Evaluation Results of Kocher Examples for Different Settings	85
7.1	Comparison of TPOD and SNI Hyperproperty	106
B.1	Path Combinations of Mis-Speculate and Taken/Not-Taken Predictors for Self-Composed Branch Instruction (correct paths are green, transient paths are red).	129



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Bibliography

- [1] Onur Aciicmez. „Yet Another MicroArchitectural Attack:: Exploiting I-Cache“. In: *Proceedings of the 2007 ACM Workshop on Computer Security Architecture. CSAW '07*. Fairfax, Virginia, USA: ACM, 2007, pp. 11–18. ISBN: 978-1-59593-890-9. DOI: 10.1145/1314466.1314469. URL: <https://doi.acm.org/10.1145/1314466.1314469>.
- [2] Onur Aciicmez, Çetin Kaya Koç, and Jean-Pierre Seifert. „On the Power of Simple Branch Prediction Analysis“. In: *Proceedings of the 2Nd ACM Symposium on Information, Computer and Communications Security. ASIACCS '07*. Singapore: ACM, 2007, pp. 312–320. ISBN: 1-59593-574-6. DOI: 10.1145/1229285.1266999. URL: <https://doi.acm.org/10.1145/1229285.1266999>.
- [3] Onur Aciicmez, Çetin Kaya Koç, and Jean-Pierre Seifert. „Predicting Secret Keys via Branch Prediction“. In: *Proceedings of the 7th Cryptographers' Track at the RSA Conference on Topics in Cryptology. CT-RSA'07*. San Francisco, CA: Springer-Verlag, 2006, pp. 225–242. ISBN: 3-540-69327-0, 978-3-540-69327-7. DOI: 10.1007/11967668_15. URL: https://dx.doi.org/10.1007/11967668_15.
- [4] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811.
- [5] Roberto Baldoni et al. „A Survey of Symbolic Execution Techniques“. In: *ACM Comput. Surv.* 51.3 (2018).
- [6] Gilles Barthe, Pedro R. D’argenio, and Tamara Rezk. „Secure Information Flow by Self-Composition“. In: *Mathematical. Structures in Comp. Sci.* 21.6 (Dec. 2011), pp. 1207–1252. ISSN: 0960-1295. DOI: 10.1017/S0960129511000193. URL: <https://doi.org/10.1017/S0960129511000193>.
- [7] Claudio Canella et al. „A Systematic Evaluation of Transient Execution Attacks and Defenses“. In: *CoRR* abs/1811.05441 (2018). arXiv: 1811.05441. URL: <https://arxiv.org/abs/1811.05441>.
- [8] Chandler Carruth. *LLVM: Introduce a new pass to do speculative load hardening to mitigate Spectre variant #1 for x86*. May 2018. URL: <https://reviews.llvm.org/D44824>.
- [9] Chandler Carruth. *Speculative Load Hardening*. Mar. 2018. URL: <https://llvm.org/docs/SpeculativeLoadHardening.html>.

- [10] Kevin Cheang et al. „A Formal Approach to Secure Speculation“. In: *Proceedings of the Computer Security Foundations Symposium (CSF)*. 2019.
- [11] Edmund Clarke, Daniel Kroening, and Flavio Lerda. „A Tool for Checking ANSI-C Programs“. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*. Ed. by Kurt Jensen and Andreas Podelski. Vol. 2988. Lecture Notes in Computer Science. Springer, 2004, pp. 168–176. ISBN: 3-540-21299-X.
- [12] Edmund M. Clarke et al. „Model Checking and the State Explosion Problem“. In: *Tools for Practical Software Verification: LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*. Ed. by Bertrand Meyer and Martin Nordio. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 1–30. ISBN: 978-3-642-35746-6. DOI: 10.1007/978-3-642-35746-6_1. URL: https://doi.org/10.1007/978-3-642-35746-6_1.
- [13] Purnendu Das. „Role of Cache Replacement Policies in High Performance Computing Systems: A Survey“. In: *Communication, Networks and Computing*. Ed. by Shekhar Verma et al. Singapore: Springer Singapore, 2019, pp. 400–410. ISBN: 978-981-13-2372-0.
- [14] Goran Doychev et al. „CacheAudit: A Tool for the Static Analysis of Cache Side Channels“. In: *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX, 2013, pp. 431–446. ISBN: 978-1-931971-03-4. URL: <https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/doychev>.
- [15] D. Evtvyushkin, D. Ponomarev, and N. Abu-Ghazaleh. „Jump over ASLR: Attacking branch predictors to bypass ASLR“. In: *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Oct. 2016, pp. 1–13. DOI: 10.1109/MICRO.2016.7783743.
- [16] Qian Ge et al. „A survey of microarchitectural timing attacks and countermeasures on contemporary hardware“. In: *Journal of Cryptographic Engineering* 8.1 (Apr. 2018), pp. 1–27. ISSN: 2190-8516. DOI: 10.1007/s13389-016-0141-6. URL: <https://doi.org/10.1007/s13389-016-0141-6>.
- [17] Corey Gough, Ian Steiner, and Winston Saunders. „CPU Power Management“. In: *Energy Efficient Servers: Blueprints for Data Center Optimization*. Berkeley, CA: Apress, 2015, pp. 21–70. ISBN: 978-1-4302-6638-9. DOI: 10.1007/978-1-4302-6638-9_2. URL: https://doi.org/10.1007/978-1-4302-6638-9_2.
- [18] Daniel Gruss and Claudio Canella. *transient.fail: Transient Execution Attacks*. Feb. 2019. URL: <https://transient.fail/>.
- [19] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. „Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches“. In: *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 897–912. ISBN: 978-1-931971-232. URL: <https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/gruss>.

- [20] Daniel Gruss et al. „Flush+Flush: A Fast and Stealthy Cache Attack“. In: *Lecture Notes in Computer Science* (2016), pp. 279–299. ISSN: 1611-3349. DOI: 10.1007/978-3-319-40667-1_14. URL: https://dx.doi.org/10.1007/978-3-319-40667-1_14.
- [21] Marco Guarnieri et al. „SPECTECTOR: Principled Detection of Speculative Information Flows“. In: *CoRR* abs/1812.08639 (2018). arXiv: 1812.08639. URL: <https://arxiv.org/abs/1812.08639>.
- [22] David Gullasch, Endre Bangerter, and Stephan Krenn. „Cache Games – Bringing Access-Based Cache Attacks on AES to Practice“. In: *Proceedings of the 2011 IEEE Symposium on Security and Privacy*. SP '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 490–505. ISBN: 978-0-7695-4402-1. DOI: 10.1109/SP.2011.22. URL: <https://doi.org/10.1109/SP.2011.22>.
- [23] Jann Horn. *speculative execution, variant 4: speculative store bypass*. Feb. 2018. URL: <https://bugs.chromium.org/p/project-zero/issues/detail?id=1528>.
- [24] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer’s Manual*. Vol. 1: Basic Architecture. 253665-060US. Sept. 2016.
- [25] Intel Corporation. *Speculative Execution Side Channel Mitigations*. 336996-003. May 2018.
- [26] Ranjit Jhala and Rupak Majumdar. „Software Model Checking“. In: *ACM Comput. Surv.* 41.4 (Oct. 2009), 21:1–21:54. ISSN: 0360-0300. DOI: 10.1145/1592434.1592438. URL: <https://doi.acm.org/10.1145/1592434.1592438>.
- [27] Paul Kocher. *Spectre Mitigations in Microsoft’s C/C Compiler*. Feb. 2018. URL: <https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html>.
- [28] Paul Kocher et al. „Spectre Attacks: Exploiting Speculative Execution“. In: *CoRR* abs/1801.01203 (2018). arXiv: 1801.01203. URL: <https://arxiv.org/abs/1801.01203>.
- [29] Esmail Mohammadian Koruyeh et al. „Spectre Returns! Speculation Attacks using the Return Stack Buffer“. In: *CoRR* abs/1807.07940 (2018). arXiv: 1807.07940. URL: <https://arxiv.org/abs/1807.07940>.
- [30] Moritz Lipp et al. „Meltdown: Reading Kernel Memory from User Space“. In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018, pp. 973–990. ISBN: 978-1-931971-46-1. URL: <https://www.usenix.org/conference/usenixsecurity18/presentation/lipp>.
- [31] Giorgi Maisuradze and Christian Rossow. „ret2spec“. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security - CCS '18* (2018). DOI: 10.1145/3243734.3243761. URL: <https://dx.doi.org/10.1145/3243734.3243761>.

- [32] Andrea Mambretti et al. *Let's Not Speculate: Discovering and Analyzing Speculative Execution Attacks*. Tech. rep. 2018.
- [33] Ross Mcilroy et al. „Spectre is here to stay: An analysis of side-channels and speculative execution“. In: *arXiv e-prints*, arXiv:1902.05178 (Feb. 2019), arXiv:1902.05178. arXiv: 1902.05178 [cs.PL].
- [34] Florian Merz, Stephan Falke, and Carsten Sinz. „LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR“. In: *Verified Software: Theories, Tools, Experiments*. Ed. by Rajeev Joshi, Peter Müller, and Andreas Podelski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 146–161. ISBN: 978-3-642-27705-4.
- [35] Microsoft. *Analysis and mitigation of speculative store bypass (CVE-2018-3639)*. May 2018. URL: <https://msrc-blog.microsoft.com/2018/05/21/analysis-and-mitigation-of-speculative-store-bypass-cve-2018-3639>.
- [36] Microsoft. *Spectre mitigations in MSVC*. Jan. 2018. URL: <https://devblogs.microsoft.com/cppblog/spectre-mitigations-in-msvc/>.
- [37] David Molnar et al. „The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks“. In: *Information Security and Cryptology - ICISC 2005*. Ed. by Dong Ho Won and Seungjoo Kim. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 156–168. ISBN: 978-3-540-33355-5.
- [38] Oleksii Oleksenko et al. „SpecFuzz: Bringing Spectre-type vulnerabilities to the surface“. In: *CoRR* abs/1905.10311 (2019). arXiv: 1905.10311. URL: <https://arxiv.org/abs/1905.10311>.
- [39] Dag Arne Osvik, Adi Shamir, and Eran Tromer. „Cache Attacks and Countermeasures: The Case of AES“. In: *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*. CT-RSA'06. San Jose, CA: Springer-Verlag, 2006, pp. 1–20. ISBN: 3-540-31033-9, 978-3-540-31033-4. DOI: 10.1007/11605805_1. URL: https://dx.doi.org/10.1007/11605805_1.
- [40] Peter Pessl et al. „DRAMA: Exploiting Dram Addressing for Cross-cpu Attacks“. In: *Proceedings of the 25th USENIX Conference on Security Symposium*. SEC'16. Austin, TX, USA: USENIX Association, 2016, pp. 565–581. ISBN: 978-1-931971-32-4. URL: <https://dl.acm.org/citation.cfm?id=3241094.3241139>.
- [41] Filip Pizlo. *What Spectre and Meltdown Mean For WebKit*. Jan. 2018. URL: <https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/>.
- [42] Fabrice Rastello. *SSA-Based Compiler Design*. 1st. Springer Publishing Company, Incorporated, 2016. ISBN: 1441962018.
- [43] J. Reineke. *Caches in WCET Analysis: Predictability - Competitiveness - Sensitivity*. 2008.

- [44] Majid Sabbagh et al. „SCADET: A Side-channel Attack Detection Tool for Tracking Prime+Probe“. In: *Proceedings of the International Conference on Computer-Aided Design. ICCAD '18*. San Diego, California: ACM, 2018, 107:1–107:8. ISBN: 978-1-4503-5950-4. DOI: 10.1145/3240765.3240844. URL: <https://doi.acm.org/10.1145/3240765.3240844>.
- [45] Andrei Sabelfeld and Andrew C. Myers. „Language-Based Information-Flow Security“. In: *IEEE Journal on Selected Areas in Communications* 21 (2003), p. 2003.
- [46] Michael Schwarz et al. „NetSpectre: Read Arbitrary Memory over Network“. In: *CoRR* abs/1807.10535 (2018). arXiv: 1807.10535. URL: <https://arxiv.org/abs/1807.10535>.
- [47] J.P. Shen and M.H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. Waveland Press, 2013. ISBN: 9781478610762. URL: <https://books.google.at/books?id=ffQqAAAAQBAJ>.
- [48] Carsten Sinz, Stephan Falke, and Florian Merz. „A Precise Memory Model for Low-Level Bounded Model Checking“. In: *Proceedings of the 5th International Conference on Systems Software Verification. SSV'10*. Vancouver, BC, Canada: USENIX Association, 2010, p. 7.
- [49] Julian Stecklina and Thomas Prescher. „LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels“. In: *CoRR* abs/1806.07480 (2018). arXiv: 1806.07480. URL: <https://arxiv.org/abs/1806.07480>.
- [50] Eran Tromer, Dag Arne Osvik, and Adi Shamir. „Efficient Cache Attacks on AES, and Countermeasures“. In: *Journal of Cryptology* 23.1 (Jan. 2010), pp. 37–71. ISSN: 1432-1378. DOI: 10.1007/s00145-009-9049-y. URL: <https://doi.org/10.1007/s00145-009-9049-y>.
- [51] Weikun Yang, Yakir Vizel, Pramod Subramanyan, Aarti Gupta and Sharad Malik. „Lazy Self-composition for Security Verification“. In: *Computer Aided Verification - 30th International Conference*. 2018, pp. 136–156. DOI: 10.1007/978-3-319-96142-2_11. URL: https://doi.org/10.1007/978-3-319-96142-2_11.
- [52] Yuval Yarom and Katrina Falkner. „FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack“. In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, 2014, pp. 719–732. ISBN: 978-1-931971-15-7. URL: <https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom>.