Towards Leakage Bounds for Side Channels based on Caches and Pipelined Executions

Technical Report

October 2021

Heiko Mantel, Alexandra Weber

Technical University of Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Modeling and Analysis of Information Systems

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - SFB 1119 - 236615297.

CROSSING

Towards Leakage Bounds for Side Channels based on Caches and Pipelined Executions

Heiko Mantel
mantel@cs.tu-darmstadt.de
Department of Computer Science,
Technical University of Darmstadt

Alexandra Weber
weber@mais.informatik.tu-darmstadt.de
Department of Computer Science,
Technical University of Darmstadt

ABSTRACT

Recent side-channel attacks like Spectre exploit the combination of multiple microarchitectural features. A prominent example is the combination of caches with instruction pipelines that optimize based on branch prediction and out-of-order execution. The combination of caches and pipelines is, e.g., exploited by Spectre v1.

Different solutions for detecting and mitigating vulnerabilities to Spectre v1 exist. In practice, not all vulnerabilities are equally serious. Different code snippets might leak different amounts of information. Mitigating each vulnerability in a program by fence insertion might be performance-wise infeasible. In addition to detection and mitigation techniques, a possibility for quantifying the security against such vulnerabilities is needed to increase the range of options. For instance, quantitative security analyses are needed to enable selective, partial mitigations that reliably establish a desired level of security against Spectre v1 vulnerabilities.

In this report, we propose a program analysis that provides quantitative upper bounds on the leakage through side channels that exploit the combination of a cache and an execution pipeline. We evaluate our analysis on multiple code snippets that might leak different amounts of information to Spectre-v1 attacks. Based on our analysis, we can successfully compare the code snippets with respect to their security impact.

1 INTRODUCTION

Side-channel attacks exploit information that a program unintentionally shares through channels that are not intended for communication. Such channels include, e.g., timing side channels [21], power side channels [23] and cache side channels [31].

The recent class of transient execution attacks [10] exploits side channels in combination with processor features that might execute instructions that would not occur in normal program execution. For instance, Spectre attacks [22] can be mounted by exploiting cache side channels in combination with instruction pipelines that speculatively execute instructions in conditional branches.

Multiple variants of transient execution attacks have been discovered in recent years, including [8, 9, 22, 27, 32, 37, 38]. One variant that is still difficult to mitigate is Spectre v1, also known as Spectre-PHT [22]. An attacker who can run a program on the same machine as the victim program and call a function from the victim program for which he supplies the parameters might be able to mount a Spectre v1 attack. In particular, a program might be vulnerable to this type of attack if it contains a snippet of the following form, where x is a parameter supplied by the attacker.

```
if (x < array1_size)
   y = array2[array1[x]];
```

The attacker might use a cache side channel to learn at which index array2 has been accessed in the second line of this snippet. The guard in the first line enforces that this index is a value from the public array1. But on a machine with an instruction pipeline, the attacker might create a situation where this index is an arbitrary value from the target program's memory. To this end, the attacker might train the branch prediction to predict that the second line will be executed. He might then call the function of the target program with an invalid value of x. In this case, the second line of the code snippet would be executed speculatively, access a secret entry from the target program's memory behind array1 and load a secret-dependent entry of array2 into the cache. When the speculative execution is rolled back, the secret-dependent entry would remain in the cache and the attacker might learn the secret information through a cache side channel.

Such vulnerabilities might lead to the leakage of arbitrary memory entries and are, thus, a serious security concern. Multiple approaches have been developed to detect whether a program might be vulnerable to Spectre attacks [3, 6, 11, 19, 20, 39]. All of these approaches focus on detecting the presence or ensuring the absence of potential leakage to Spectre attacks.

If a potential leakage is detected, one has the choice to accept the leakage or to mitigate the leakage, e.g., by inserting fence instructions that prevent speculative execution. Given that fence instructions incur a significant performance cost and that vulnerable code snippets might be more complex than the simple example shown above, more flexibility is needed.

Consider, for instance, code snippets of the following forms:

```
if (x < a1_size)          if (x < a1_size)
   y = a2[a1[x]&0xF];        y = a2[a1[x]>>15];
```

Such snippets might leak information from the memory beyond a1 in the same way as the basic snippet shown before. The amount of information leaked might, however, be smaller than the amount leaked by the basic snippet, because the secret is modified (shifted or masked) before it is transmitted through the cache side channel.

In order to compare different implementation alternatives with respect to the extent of their vulnerability to Spectre-v1-like attacks, a security analysis that only detects the presence of vulnerabilities is not fine-grained enough. A security analysis that provides reliable quantitative results is needed in addition. Such quantitative results would also be useful to decide whether a potentially expensive hardening with fence insertion is needed, or to compare the success of different non-conservative fence-insertion techniques.

Approaches to compute reliable upper leakage bounds already exist for cache side channels [5, 15, 16, 29, 43]. These approaches

focus on cache side channels in isolation and do not take into account instruction pipelining and resulting transient executions.

In this report, we propose a program analysis that computes upper bounds on the side-channel leakage with respect to attacks that, like Spectre v1, exploit the combination of a cache and an instruction pipeline. Technically, our program analysis computes bounds with respect to information-theoretic notions of leakage through an abstract reachability analysis.

To define a reliable and suitably precise abstract reachability analysis, we need a formal model of execution as a reference point. Existing execution models that are used as reference points for quantitative program analyses, including [5, 15, 16, 29, 43], do not take into account instruction pipelines. To overcome this limitation, we develop a formal model of execution for a simple assembly language that covers both, a basic fully-associative cache and a basic five-stage instruction pipeline with static branch prediction and out-of-order execution.

The main insight from our evaluation is that the leakage bounds computed with our analysis provide a good guidance how the security level of different code snippets compares. Our evaluation covers three variations of code snippets that might occur in real-world implementations. The leakage bounds we obtain vary significantly across these code snippets. For the examples that we consider, the leakage bounds range from 4bit (for a snippet variant that uses a masked value to index an array access) to 32bit (for the simplest type of snippet that might leak an entire memory entry). That is, our analysis allows us to compare different code variants with respect to their level of security bounds against Spectre-v1-like attacks.

In summary, the main contributions of this report are:

- a model of execution that takes into account both, a cache and an instruction pipeline, and that can be used as a concrete domain in a quantitative program analysis,
- a program analysis that computes upper bounds on the leakage of programs through the vulnerabilities that arise from the combination of cache side channels with instruction pipelining, and
- a comparison of the security level across three vulnerable code snippets, which is based on quantitative leakage bounds obtained with our program analysis.

The rest of this report is structured as follows. In Section 2 we summarize the basic notions related to caches, instruction pipelines, and Spectre v1 attacks and the concepts from information theory and abstract interpretation that we base on. In Section 3, we describe our formal model of execution. In Section 4, we describe our program analysis. In Section 5, we evaluate our analysis and use it to compare the security level across different code snippets. We discuss related work in Section 6 and conclude in Section 7.

2 PRELIMINARIES

2.1 Relevant notions of computer architecture

2.1.1 Caches and cache side channels. Caches are hardware components that store selected entries from the main memory closer to the CPU. If the CPU needs to access a memory entry, it first checks whether the entry is available in the cache. In this case, the entry can be retrieved from the cache quickly (*cache hit*). Otherwise, the entry has to be retrieved from the main memory, which takes significantly longer (*cache miss*). In the latter case, the entry is then added to the cache for future accesses. Modern computer architectures usually use a hierarchy of multiple caches with increasing size.

Caches are organized in so-called *cache sets*. Each memory entry can be cached in only one of the cache sets. This set is determined based on the address of the entry in memory. Each cache set can hold multiple memory entries in so-called *cache lines*. The number of cache lines per cache set is called associativity. Caches that only consist of one cache set are called *fully associative*. When a memory entry is added to a cache set in which all cache lines are already occupied, an existing entry from the cache is *evicted*. There are multiple strategies for eviction. For instance, the *FIFO* strategy removes the cache entry that has been cached for the longest time.

In a cache-side-channel attack, attackers exploit that a target program has unintentionally encoded secret information into the state of a shared cache. For instance, if a program accesses an array at a secret-dependent index, the memory entry that is loaded into the cache might also depend on the secret. There are multiple ways in which an attacker could exploit this. We focus on access-based attacks in this report. Two possible techniques for such attacks are Evict+Time [30] and Flush+Reload [45]. In an Evict+Time attack, the attacker evicts selected cache entries and times the victim execution to detect whether the victim accesses the evicted entries. In a Flush+Reload attack, the attacker uses a *flush* instruction to remove an entry from the cache, lets the victim execute, and then reloads the flushed line to see whether it has already been added to the cache again by the victim.

2.1.2 Instruction pipelines and transient-execution vulnerabilities. Instruction pipelines parallelize the execution of multiple instructions within a single CPU. To this end, the execution of instructions is split into multiple stages, e.g., the fetching the instruction (*fetch* stage), dispatching the instruction to functional unit of the CPU that can perform the operation (*dispatch* stage), executing the instruction (*execute* stage), and committing the result of the instructions to the destination registers (*commit* stage). To execute instructions in parallel, the fetch stage already fetches the next instruction once the previous instruction has advanced into the dispatch stage, etc.

There are cases in which this parallel processing would have to slow down. In case a conditional jump instruction is fetched, the result of the previous instruction that computes the condition might not be available. In order to avoid that the next instruction fetch has to wait for the result (*pipeline stalling*), modern CPUs use branch prediction to guess which instruction to fetch next [35]. The corresponding instruction is then fetched into the pipeline and executed speculatively. If it turns out that the guess was incorrect and the actual branching decision is a different one, the speculative execution is rolled back. In particular, the intermediate results are discarded, the speculatively executed instructions are removed (*flushed*) from the pipeline and the correct instruction gets fetched.

In case an instruction in the execute stage depends on memory operands, these operands might take longer to retrieve because they are not available in the cache. To avoid pipeline stalling in this case, modern CPUs use out-of-order execution in the execute stage [36]. To this end, instructions get dispatched into a buffer

called *reservation station*. Instructions from this buffer can be executed out of order if they do not depend on earlier instructions in the buffer. After the execution, the results are written into a *reorder buffer* in which they are available to the other instructions in the execution phase. From this reorder buffer, the results are committed to the actual registers in the original order of the instructions.

The combination of branch prediction and out-of-order execution can lead to transient-execution vulnerabilities. While the pipeline and reorder buffer are cleared in case of a mispredicted branching decision, other parts of the microarchitectural state are not cleared. In particular, the cache is not flushed when a rollback occurs [22]. That is, if a secret-dependent memory access is executed speculatively (e.g., for the snippets from Section 1), the resulting secret-dependent cache entry remains. An attacker might extract the secret using techniques like Evict+Time or Flush+Reload.

2.2 Concepts for side-channel quantification

2.2.1 Information theory. To model the leakage of programs in Spectre-v1-like attacks, we rely on concepts from information theory. We model the possible program inputs by a random variable $\mathcal{I}$ with the set I of possible values and with prior distribution $\vec{\pi}$. We model the possible cache-side-channel observations by a random variable $\mathcal{O}$ with possible values from the set O. We view the program as an information-theoretic channel $C : I \to O$.

The attacker's initial uncertainty about the program input is captured by the min-entropy $H_\infty(\mathcal{I}) = -\log_2 \max_i \pi_i$ [34]. His remaining uncertainty after observing the cache is captured by the conditional min-entropy $H_\infty(\mathcal{I}|\mathcal{O}) = -\log_2 \Sigma_j p(o_j) \cdot \max_i \frac{p(o_j|i_i)\cdot p(i_i)}{p(obs_j)}$, where $p(o_j)$ is the probability for observation o_j, $p(i_i) = \pi_i$ and $p(o_j|i_i)$ is the probability for observation o_j if the input is i_i.

We capture the side-channel leakage as the decrease in min-entropy through the side-channel observation, namely $L_\infty(\vec{\pi}, C) = H_\infty(\mathcal{I}) - H_\infty(\mathcal{I}|\mathcal{O})$. This leakage is upper-bounded by the logarithm of the possible values of $\mathcal{O}$, i.e., $L_\infty(\vec{\pi}, C) \leq \log_2 |O|$ [25].

2.2.2 Abstract reachability analysis. Directly computing $\log_2 |O|$ would require to know the set O, i.e., all side-channel observations that are reachable for any possible program input from I. Since computing O is infeasible for non-trivial programs, we compute an overapproximation $O' \supseteq O$ instead and compute $\log_2 |O'|$, which is also a leakage bound due to the monotonicity of $\log_2$. We compute the overapproximation using abstract interpretation [13].

To this end, we first define a model of execution as a reference point for our computations. More concretely, we define a concrete domain $\mathcal{D}$ of possible configurations during the execution of a program, and a concrete semantics $\mathsf{upd}_\mathcal{D} : \mathcal{D} \to \mathcal{D}$, which captures the effect of executing a program for one clock cycle.

We then define an abstract domain $\overline{\mathcal{D}}$ of abstract program configurations, where $\alpha : \mathcal{P}(\mathcal{D}) \to \overline{\mathcal{D}}$ is an abstraction function that maps each set of concrete configurations to an abstract configuration that represents all concrete configurations from the set. The concretization function $\gamma : \overline{\mathcal{D}} \to \mathcal{P}(\mathcal{D})$ maps each abstract configuration to the set of all concrete configurations that it represents. We define an abstract semantics $\mathsf{upd}_{\overline{\mathcal{D}}} : \overline{\mathcal{D}} \to \overline{\mathcal{D}}$. This abstract semantics reliably overapproximates the concrete semantics if $\forall D \subseteq \mathcal{D}.\ \{d' \mid \exists d \in D.\ d' = \mathsf{upd}_\mathcal{D}(d)\} \subseteq \gamma(\mathsf{upd}_{\overline{\mathcal{D}}}(\alpha(D)))$.

To analyze a program, we first compute an abstract configuration that represents the set of concrete configurations that capture all possible initial states of the program execution for all possible inputs. We then compute a fixed point of the function $\mathsf{upd}_{\overline{\mathcal{D}}}$ to obtain an abstract configuration that represents a superset of all possible final concrete configurations. Finally, we compute the set O' of cache-side-channel observations arising from this superset of possible final configurations and obtain the leakage bound $\log_2 |O'|$.

This combination of information theory and abstract reachability analysis for cache-side-channel quantification is a popular approach in scenarios without pipelining [5, 15, 16, 24, 28, 29, 43]. In this report, we apply this approach in a scenario with pipelining. To our knowledge, we are the first to develop a program analysis that computes reliable upper bounds on the cache-side-channel leakage of programs that are executed in a pipelined fashion.

2.3 Notational Conventions

We denote the domain and the range of a function f by $\mathtt{dom}(f)$ and $\mathtt{rng}(f)$, respectively. We use $X \rightharpoonup Y$ to denote the set of all partial functions from elements of X to elements of Y. If a partial function f is undefined on value x, i.e., $x \notin \mathtt{dom}(f)$, we write $f(x)\uparrow$.

By $X^\star$, we denote the set of all sequences of elements from X. We denote the concatenation of two sequences τ and τ' by $\tau \bullet \tau'$, the length of the sequence τ by $|\tau|$ and the i-th element of τ by $\tau[i]$.

We write $\mathcal{P}(X)$ for the powerset of X.

3 MODEL OF EXECUTION

We model executions on architectures with a cache and a pipeline for a simple assembly language that we call *p*ASM.

3.1 Syntax

We use a context-free grammar in Extended Backus-Naur Form (EBNF) to define the syntax of *p*ASM.

Definition 1. The *set of assembly instructions* is the set *Insts* of all words that can be produced by the production rules for the non-terminal symbol *I* in the following EBNF. Analogoulsy,

- the *set of registers* is the set *Regs* of all words for *R*,
- the *set of 32-bit values* is the set *Vals* of all words for *V*,
- the *set of data addresses* is the set *Dadds* of all words for *DA*,
- the *set of instruction addresses* is the set *Iadds* of all words for *IA* in the following EBNF.

```
I  =  'mov-rc' R V | 'mov-rm' R DA R | 'and-rm' R DA R |
      'shr-rm' R DA R | 'jge' IA R R | 'fence' | 'nop'.
R  =  'eax' | 'ebx' | 'ecx'.
V  =  32 * ('0'|'1').
DA =  'd' '1'-'9' {'0'-'9'}.
IA =  'i' '1'-'9' {'0'-'9'}.
```

We call the first terminal-symbol in each instruction the instruction's *mnemonic*. To improve readability, we omit the leading zeroes for values from the set *Vals*.

Intuitively, instructions with mnemonic **mov-rc** move a value given in 32-bit binary representation into a register. For instance, **mov-rc** *eax* 10 moves the value 2 into register *eax*. Instructions with mnemonic **mov-rm** retrieve a value from a data address given by a base address and an offset stored in a register and move the

value into another register. For instance, if the register *eax* contains value 4, the instruction **mov-rm** *ebx d*16 *eax* would move the value stored at data address *d*20 to the register *ebx*. The instruction **and-rm** *ebx d*16 *eax* would retrieve the value from data address *d*20, compute the bit-wise AND with the value stored in register *ebx* and store the result in register *ebx*. Analogously, **shr-rm** *ebx d*16 *eax* would retrieve the value from data address *d*20, shift it to the right by the offset stored in register *ebx* and store the result in register *ebx*. Instructions with the mnemonic **jge** jump to a specified instruction address in case the value stored in the first operand register is greater than or equal to the value stored in the second operand register. For instance **jge** *i*7 *eax ebx* would jump to instruction address *i*7 if the value in *eax* is greater than or equal to the value in *ebx*. Finally, instructions with mnemonic **fence** prevent speculative execution and instructions with mnemonic **nop** perform no operation at all.

Note that, we consider infinite but disjoint sets of data addresses and instruction addresses. Since we focus on cache side channels in this report, we assume that buffer overflows and similar errors have already been eliminated by a separate analysis.

We use the functions $\texttt{iadd} : \mathit{Iadds} \times \mathbb{N} \to \mathit{Iadds}$ and $\texttt{dadd} : \mathit{Dadds} \times \mathbb{N} \to \mathit{Dadds}$ to increment instruction and data addresses, respectively, by a given natural number.

Definition 2. The *set of programs in pASM* is the set $\mathit{Programs} = \mathit{Iadds} \rightharpoonup \mathit{Insts}$ and the *set of well-formed programs in pASM* is the set $\mathit{Progs}_{wf} = \{pr \in \mathit{Programs} \mid wf(pr)\}$, where the well-formedness predicate $wf(pr)$ is the conjunction of the following three properties.

- $i0 \in \mathsf{dom}(pr)$,
- $\forall n \in \mathbb{N}.\ in \in \mathsf{dom}(pr) \Rightarrow in - 1 \in \mathsf{dom}(pr)$, and
- $\forall ia, ia' \in \mathit{Iadds}.\ \forall r, r' \in \mathit{Regs}.\ pr(ia) = \textbf{jge}\ ia'\ r\ r' \Rightarrow ia' \in \mathsf{dom}(pr) \wedge ia' > ia$.

Intuitively, a program is a partial mapping from consecutive instruction addresses to instructions, such that all targets of jump instructions are within the program and all jumps are forward.

Example 3.1 (Simple Code Snippet for Spectre v1). Let *d*1 be the base address of an array *a*1 that spans 3 memory entries. Let *d*6 be the base address of a second array *a*2. Let *d*0 be the data address at which the size of the first array is stored. The following program `p-simple` ∈ *Programs* (with parameter in *eax*) is an example of a basic code snippet that might be vulnerable to Spectre v1.

Listing 1: `p-simple`

```
1  mov-rc ebx 0            // initialize ebx
2  mov-rc ecx 1111         // initialize ecx
3  mov-rm ebx d0 ebx       // load size of a1 into ebx
4  jge i7 eax ebx          // jump to i7 if eax >= ebx
5  mov-rm ecx d1 eax       // ecx = a1[eax]
6  mov-rm ecx d6 ecx       // ecx = a2[a1[eax]]
7  nop
```

◊

The program `p-simple` is a *p*ASM variant of the simple code snippet from Section 1. It uses the attacker-controlled parameter in register *eax* as an index to access array *a*1 in instruction *i*5. It then encodes the memory entry obtained with this access into the cache by the access to *a*2 in instruction *i*6. Instruction *i*4 checks whether the value in *eax* is within bounds for array *a*1.

Note that, `p-simple` is well-formed, i.e., $\texttt{p-simple} \in \mathit{Progs}_{wf}$.

Example 3.2 (Code Snippet Variants for Spectre v1). Let *d*1, *d*2 and *d*0 be as in Example 3.1. The are examples of *p*ASM programs that might be vulnerable to Spectre v1 attacks.

Listing 2: `p-mask`

```
1  mov-rc ebx 0
2  mov-rc ecx 1111
3  mov-rm ebx d0 ebx
4  jge i7 eax ebx
5  and-rm ecx d1 eax
6  mov-rm ecx d6 ecx
7  nop
```

Listing 3: `p-shift`

```
1  mov-rc ebx 0
2  mov-rc ecx 1111
3  mov-rm ebx d0 ebx
4  jge i7 eax ebx
5  shr-rm ecx d1 eax
6  mov-rm ecx d6 ecx
7  nop
```

◊

The programs `p-mask` and `p-shift` are *p*ASM variants of the second and third code snippet from Section 1. Like `p-simple`, they access array *a*1 at the attacker-controlled index in *eax*. They then encode information that depends on the result of this memory access into the cache in Instruction *i*6.

Program `p-mask` encodes the last 4 bit of the result into the cache, because it applies the mask 0b1111, stored in *ecx* in Instruction *i*5. Analogously, the program `p-shift` encodes the all bits except for the last 15 bit into of the result into the cache, because it applies a shift by the offset 0b1111, stored in *ecx* in Instruction *i*5.

If the memory entry accessed by Instruction *i*5 is secret (i.e., if the access to *a*1 is out of bounds due to branch misprediction), all three programs, `p-mask`, `p-shift` and `p-simple` leak secret information to the cache. However, the amount of information that they leak differs significantly across the three programs.

3.2 Semantics

We model the semantics of *p*ASM with respect to an execution environment that features registers, an instruction memory, a main memory for data, and a data cache. Recall from Section 3.1 that we consider three registers, *eax*, *ebx* and *ecx*, and disjoint but infinite data and instruction memories. For simplicity, we consider one level of fully-associative cache with 512 cache lines of size 32bit. This simplification allows us to focus on the interplay between the cache and instruction pipeline. The effects of different cache parameters (e.g., associativity, cache size, ...) on side-channel leakage have already been investigated at the example of side channels in AES implementations, e.g., in [16, 29].

The execution environment features an instruction pipeline with branch prediction and out-of-order execution. We consider a static branch-prediction strategy that requires no additional architectural components. For out-of-order execution, the environment features two additional components: reservation stations and a reorder buffer. More concretely, we consider an architecture with four reservation stations and four positions in the reorder buffer.

An overview of all components we consider is shown in Figure 1.

3.2.1 Configurations. We capture the state of each component at a given point during program execution by a configuration.

Memory and Registers. For the instruction memory, the set of possible configurations is given by the possible mappings from instruction addresses to instructions in the set Progs_{wf} of well-formed programs. For the data memory and registers, we define the

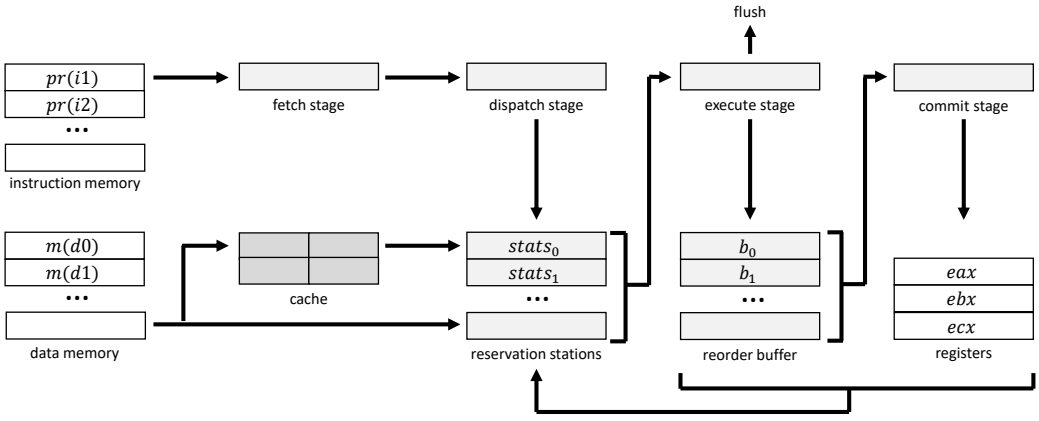

Figure 1: Overview of execution model

sets of possible configurations, respectively, by the sets of functions $RStates = Regs \to Vals$ and $MStates = Dadds \to Vals$.

Cache. We define the set of possible cache configurations by $CStates = Dadds \to \mathbb{N}$. A cache configuration captures a snapshot of the cache by mapping each data address to the index of the cache line in which the respective memory entry is stored. Values that are greater than or equal to 512 capture the case in which a memory entry is not stored in the cache. This faithfully captures the possible states of a 2-KiB fully associative cache with 32-bit cache lines.

Pipeline. We capture the possible states of the pipeline during program execution as triples of the configurations of the components that belong to the pipeline: the pipeline stages, the reservation stations, and the reorder buffer.

The set of possible *pipeline-stage configurations* is the set of functions $StStates = Stages \to StConts$. A pipeline-stage configuration maps pipeline stages from the set $Stages = \{fet, dis, exe, com\}$ to pipeline contents from the set $StConts = Iadds \cup \{\bot, \top\}$. Each such configuration covers four pipeline stages: fet (fetch stage), dis (dispatch stage), exe (execute stage), and com (commit stage). That is, it captures the four basic stages needed for out-of-order execution. It maps each pipeline stage to the address of the instruction that is currently processed at this pipeline stage. The cases in which a pipeline stage is idle are captured by the symbols $\top$ and $\bot$. The symbol $\top$ captures the case that a pipeline stage is stalled on purpose (due to a fence instruction. The symbol $\bot$ captures the case that a pipeline stage is stalled because no instruction is ready to be processed (e.g., due to a pipeline flush or due to dependencies).

The set of possible *reservation-station configurations* is the set $RSStates = RSConts^\star$. A reservation-station configuration captures the list of instructions in the reservation stations. Each entry in the list is a triple from the set $RSConts = Iadds \times MLists \times RLists$. The first element of the triple is the address of the instruction to be executed. The second and third element capture the conditions that must be fulfilled before executing it. Before executing an instruction, all memory entries must be retrieved from the memory and the dependencies of all register operands must be resolved, i.e., the correct value for the register must be available in the reorder buffer. The second element, the *memory-access list* from the set $MLists = (Dadds \times \mathbb{N})^\star$ captures the remaining time required to retrieve each memory operand. It is a list of pairs, where each pair consists of a data address and a natural number that captures the number of clock cycles remaining until the data from the address is available. The third element, the *register-dependence list* from the set $RLists = (Regs \times Iadds)^\star$, captures the dependencies of the register operands. It is a list of pairs, where each pair consists of a register and the address of the last instruction on which the register's value depends.

The set of possible *reorder-buffer configurations* is $BStates = BConts^\star$. A reorder-buffer configuration captures the list of register updates in the reorder buffer. Each entry in the list is a triple from the set $BConts = Iadds \times Regs \times Vals$. The triple consists of the address of the instruction that triggered the update, the register to be updated and the value to which the register shall be updated.

The intuition behind the memory-access list in a reservation station is as follows. For entries that are in the cache, the remaining cycles start at a value t_{hit} that captures the cycles a memory access costs in case of a cache hit. For entries that are not in the cache, the remaining cycles start at t_{miss}. The value t_{miss} captures the additional time needed to access the main memory and add the missing entry to the cache. For each execution step, the remaining cycles decrease. Once the remaining cycles reach t_{hit}, the entry is added to the cache. This intuition will be formalized in Section 3.2.2.

Overall, we capture the possible pipeline states by the set

$$PStates = StStates \times RSStates \times BStates$$

of *pipeline configurations*. We use the functions $\texttt{stages}(pi)$, $\texttt{stat}(pi)$, and $\texttt{buf}(pi)$ to extract the pipeline-stage configuration, reservation-station configuration and the reorder-buffer configuration from a pipeline configuration pi, respectively. We use the functions $\texttt{fstage}(pi)$, $\texttt{dstage}(pi)$, $\texttt{estage}(pi)$, and $\texttt{cstage}(pi)$ to extract the contents of each pipeline stage from a pipeline configuration. To extract the instruction address from a reservation-station or reorder-buffer configuration, we use the function $\texttt{addr}((ia, x, y)) = ia$.

Putting all together. Overall, we model snapshots of program execution in the execution environment as tuples that combine the configurations of all architectural components.

Definition 3. The *set of all concrete configurations* is the set

$$States = Progs_{wf} \times RStates \times MStates \times CStates \times PStates.$$

By the definition of the sets of component configurations, the set *States* faithfully captures the possible snapshots of executions in an environment that processes 32-bit values and consists of disjoint, infinite instruction and data memories, three registers *eax*, *ebx* and *ecx*, a fully-associative 2KiB cache with 32bit line size, and a basic four-stage pipeline that supports out-of-order execution by reservation stations and a reorder buffer.

Next, we define which configurations from *States* are actually reachable in executions of well-formed *p*ASM programs.

3.2.2 Execution Steps. We define the semantics of *p*ASM in a small-step fashion. More concretely, we define a function that we call the concrete update function and that captures the changes to a given configuration after one clock cycle in the execution. To this end, we first introduce auxiliary functions that capture the changes to individual components of a configurations and then combine them to the overall function.

Memory and Registers. Since *p*ASM does not have any instructions that write to memory, the concrete update functions for the instruction and data memory are simply identity functions.

Definition 4. The *concrete instruction-memory update* is the function $\texttt{c-upd}_{\text{pr}} : Progs_{wf} \to Progs_{wf}$, s.t. $\texttt{c-upd}_{\text{pr}}(pr) = pr$.

Definition 5. The *concrete data-memory update* is the function $\texttt{c-upd}_{\text{mem}} : MStates \to MStates$, s.t. $\texttt{c-upd}_{\text{mem}}(m) = m$.

The new values of registers after an execution step are based on the pending commits in the reorder buffer. Thus, we define the concrete update function based on the reorder buffer.

Definition 6. The *concrete register update* is the function $\texttt{c-upd}_{\text{reg}} : RStates \times BStates \times StConts \to RStates$, such that

$$\texttt{c-upd}_{\text{reg}}(rs, b, ia) = \left(\lambda r.\ \begin{cases} v & \text{if } \exists i < |b|.\ b_i = (ia, r, v) \\ rs(r) & \text{otherwise} \end{cases}\right).$$

Given the prior register configuration *rs*, the reorder-buffer configuration *b* and the instruction address *ia* of the instruction that is currently processed by the commit stage, $\texttt{c-upd}_{\text{reg}}(rs, b, ia)$ captures the effect of the commit stage on the register configuration.

If the instruction that is currently processed by the commit stage triggered an entry in the reorder buffer, the entry is committed to the corresponding register. All other registers remain unchanged. If the commit stage is idle ($\bot$ or $\top$) or if the instruction processed by the stage triggered no buffer entry, all registers remain unchanged.

Note that, the function is well-defined for configurations that result from executions of well-formed programs. Since each instruction is executed at most once (no backward jumps) and each instruction writes at most to one register, there is at most one entry in the reorder buffer for each instruction.

Cache. We capture the effect of memory accesses in the reservation stations on the cache configuration by a function.

Definition 7. The *concrete cache update* is the function $\texttt{c-upd}_{\text{cache}} : CStates \times RSStates \to CStates$, s.t.

$$\texttt{c-upd}_{\text{cache}}(c, stats) = \left(\lambda d.\ \begin{cases} 0 & \text{if } \exists ml \in MLists.\ \exists rl \in RLists. \\ & \quad \exists i < |stats|.\ \exists j < |ml|. \\ & \quad\quad stats_i = (ia, ml, rl) \wedge \\ & \quad\quad ml_j = (d, t_{hit} + 1) \\ c(d) + 1 & \text{if } \exists ml \in MLists.\ \exists rl \in RLists. \\ & \quad \exists i < |stats|.\ \exists j < |ml|. \\ & \quad \exists d' \in Dadds. \\ & \quad\quad stats_i = (ia, ml, rl) \wedge \\ & \quad\quad ml_j = (d', t_{hit} + 1) \wedge \\ & \quad\quad d' \neq d \wedge c(d') \geq 512 \\ c(d) & \text{otherwise} \end{cases}\right)$$

The concrete cache update captures the effect of the memory accesses of instructions in the reservation station on the cache. Given the prior cache *c* and the reservation-station configuration *stats*, the cache resulting from a concrete cache update maps the given address *d* to 0 if there is a reservation station whose memory-access list contains the pair $(d, t_{hit} + 1)$. If the reservation stations contain a pair $(d', t_{hit} + 1)$ for an address that is not cached previously and that is different from *d*, the updated cache increases the cache position of *d* by 1. This captures the FIFO replacement policy. Finally, if no memory access with $t_{hit} + 1$ remaining cycles occurs in the reservation stations, the cache remains unchanged.

The function is well-defined for configurations that result from the execution of well-formed programs. Each instruction in *p*ASM accesses at most one memory entry. Therefore, there is at most one access in the reservation stations with $t_{hit} + 1$ remaining cycles.

Pipeline. The concrete update function for pipeline configurations is based on concrete update functions for the pipeline stages, the reservation stations, and the reorder buffer.

Pipeline Stages. During program execution, the individual instructions pass through the pipeline stages, such that each stage is updated to the instruction that was processed by the preceding stage beforehand. The fetch and execute stages are special cases. The former depends on the branch prediction and the latter depends on out-of-order execution.

As mentioned above, we assume a static branch-prediction strategy. More concretely, we model a strategy that always predicts that branches are not taken. To model out-of-order execution, we assume that the first instruction from the reservation stations that is ready for execution is the next instruction to be executed. To this end, we define what it means to be ready for execution by the function $\texttt{ready} : RSConts \times RSStates \to \mathbb{B}$, such that

$$\begin{aligned} &\texttt{ready}((ia, ml, rl), stats) = \\ &\quad (\forall i < |ml|.\ \exists d \in Dadds.\ ml_i = (d, 0)) \wedge \\ &\quad (\forall j < |rl|.\ \exists r \in Regs.\ \exists ia' \in Iaddrs. \\ &\qquad rl_j = (r, ia') \wedge (\forall k < |stats|.\ \texttt{addr}(stats_k) \neq ia'))). \end{aligned}$$

Given the entry (ia, ml, rl) of a reservation station and a list *stats* of other reservation-station entries, the predicate captures that all

memory entries from the memory-access list ml are available (0 clock cycles remain) and the instructions on which the execution of ia depends (stored in the register-dependence list rl) are not in the list $stats$. The idea is to use this predicate with a parameter $stats$ that captures the entries preceding (ia, ml, rl) in the reservation stations. When used in this way, the predicate captures that all memory-entries required to execute ia are available and all instructions on which ia depends have been executed already.

We use the predicate in this way to determine the next ready instruction in a reservation-station configuration by the function $\mathtt{next} : RSStates \times RSStates \rightharpoonup StConts$, such that $\mathtt{next}(stats, prevs) =$

$$\begin{cases} \uparrow & \text{if } stats = \langle\rangle \\ \mathtt{addr}(rc) & \text{if } stats = \langle rc\rangle \bullet stats' \wedge \\ & \quad \mathtt{ready}(rc, prevs) \\ \mathtt{next}(stats', prevs \bullet \langle rc\rangle) & \text{if } stats = \langle rc\rangle \bullet stats' \wedge \\ & \quad \neg\mathtt{ready}(rc, prevs). \end{cases}$$

Given the reservation-station configuration $stats$, $\mathtt{next}(stats, \langle\rangle)$ is either the first instruction from $stats$ that satisfies ready, or (if no instruction is ready) the function is undefined.

We now define the concrete update function for pipeline stages.

Definition 8. The *pipeline-stage update* is the function $\mathtt{c\text{-}upd}_{\mathtt{st}} : PStates \times Progs_{wf} \rightharpoonup StStates$, such that $\mathtt{c\text{-}upd}_{\mathtt{st}}(pi, pr) =$

$$\left(\lambda s.\ \begin{cases} \mathtt{iadd}(\mathtt{fstage}(pi), 1) & \text{if } s = fet \wedge \\ & \quad \mathtt{fstage}(pi) \in Iadds \wedge \\ & \quad \neg pr(\mathtt{fstage}(pi)) = \textbf{fence} \wedge \\ & \quad \neg pr(\mathtt{iadd}(\mathtt{fstage}(pi), 1))\uparrow \\ \top & \text{if } s = fet \wedge \\ & \quad (\mathtt{fstage}(pi) = \top \vee \\ & \quad\ (\mathtt{fstage}(pi) \in Iadds \wedge \\ & \quad\quad pr(\mathtt{iadd}(\mathtt{fstage}(pi), 1))\uparrow)) \\ \bot & \text{if } (s = fet \wedge \\ & \quad\ (pr(\mathtt{fstage}(pi)) = \textbf{fence} \vee \\ & \quad\ (\mathtt{fstage}(pi) = \bot \wedge \\ & \quad\ (pr(\mathtt{dstage}(pi)) = \textbf{fence} \vee \\ & \quad\ \exists ia \in Iadds.\ \exists i < |\mathtt{stat}(pi)|. \\ & \quad\quad ia = \mathtt{addr}(\mathtt{stat}(pi)_i) \wedge \\ & \quad\quad pr(ia) = \textbf{fence}))))) \vee \\ & \quad (s = exe \wedge \\ & \quad\ \mathtt{next}(\mathtt{stat}(pi), \langle\rangle)\uparrow) \\ \mathtt{fstage}(pi) & \text{if } s = dis \\ \mathtt{next}(\mathtt{stat}(pi), \langle\rangle) & \text{if } s = exe \wedge \neg\mathtt{next}(\mathtt{stat}(pi), \langle\rangle)\uparrow \\ \mathtt{estage}(pi) & \text{if } s = com. \end{cases}\right)$$

The first stage of the pipeline, the fetch stage, retrieves the next instruction to be loaded into the pipeline. If the previous instruction in the fetch stage was a regular instruction, i.e., no **fence** and no stalling, and the program has not been fetched completely yet, the next instruction is retrieved from the subsequent instruction address (Case 1). This is also the case for **jge** instructions, since we model a static branch-prediction strategy, which always predicts that branches will not be taken.

If the previous contents of the fetch stage was $\top$ (i.e., the program has been fetched completely), the fet stage remains at $\top$ (Case 2).

If the previous instruction in the fetch stage was a **fence** or if a previous **fence** has not yet reached the exe stage, the pipeline stalls. This is captured by the symbol $\bot$ (Case 3).

The dispatch stage is updated to the previous contents of the fetch stage (Case 4), the execute stage is updated to the next ready instruction from the reservation stations (Case 5) or stalls in case no instruction is ready (Case 3), and the commit stage is updated to the previous contents of the execute stage (Case 6).

So far, we have captured the regular case in which no branch misprediction occurs. In case a branch misprediction is detected, the pipeline is not updated regularly, but is flushed.

A misprediction is recognized based on the actual operand values of a conditional jump instruction when it is about to be executed. Due to out-of-order execution, the actual values for register operands might be located either in the reorder buffer or in the actual registers. We capture the retrieval of the values by the function $\mathtt{getr} : Regs \times RStates \times BStates \rightarrow Vals$, s.t.

$$\mathtt{getr}(r, rs, b) = \begin{cases} rs(r) & \text{if } b = \langle\rangle \\ \mathtt{getr}(r, rs, b') & \text{if } \exists ia \in Iadds.\ \exists r' \in Regs. \\ & \quad b = b' \bullet \langle(ia, r', v)\rangle \wedge \\ & \quad r' \neq r \\ v & \text{if } \exists ia \in Iadds. \\ & \quad b = b' \bullet \langle(ia, r, v)\rangle. \end{cases}$$

The register-value function $\mathtt{getr}(r, rs, b)$ first searches the reorder buffer b for the last update to the register r. If such an update is found, the respective value is retrieved. If no such update is found, the value is retrieved from the register configuration rs. Since entries are added to the buffer in program order (see Definition 11), this always results in the correct register value.

Based on getr, we capture whether a pipeline flush happens by $\mathtt{isfl} : PStates \times Progs_{wf} \times RStates \rightarrow \mathbb{B}$, such that

$$\begin{aligned} \mathtt{isfl}(pi, pr, rs) = \exists ia \in Iadds.\ \exists r, r' \in Regs. \\ pr(\mathtt{estage}(pi)) = \textbf{jge}\ ia\ r\ r' \wedge \\ \mathtt{getr}(r, rs, \mathtt{buf}(pi)) \geq \mathtt{getr}(r', rs, \mathtt{buf}(pi)). \end{aligned}$$

That is, a pipeline flush happens if the execute stage is processing a **jge** instruction that would trigger a jump, i.e., the value of the first operand register is greater than or equal to the value of the second one (recall the always-not-taken branch-prediction strategy).

Definition 9. The *pipeline flush function* is the function $\mathtt{flush} : PStates \times Progs_{wf} \rightarrow StConts$, such that

$$\mathtt{flush}(pi, pr) = \left(\lambda s.\ \begin{cases} ia & \text{if } s = fet \wedge \exists r, r' \in Regs. \\ & \quad pr(\mathtt{estage}(pi)) = \textbf{jge}\ ia\ r\ r' \\ \top & \text{otherwise} \end{cases}\right).$$

In case of a pipeline flush, the fetch stage is set to the correct jump target and all other pipeline stages are set to $\top$, i.e., idle.

Reservation Stations. One clock cycle during program execution affects the reservation stations in three ways: a new instruction is dispatched to the reservation stations, the ongoing retrieval of memory operands for the instructions in the stations progresses, and the next ready instruction (if any) moves to the execute stage and is removed from the reservation stations.

We capture the dispatch of a new instruction by $\mathtt{dodisp} : StConts \times Progs_{wf} \times CStates \times RStates \times \times BStates \to RSStates$, such that

$$\mathtt{dodisp}(sc, pr, c, rs, b) =$$

$$\begin{cases} \langle\rangle & \text{if } sc \notin Iadds \\ \langle(sc, \langle\rangle, \langle\rangle)\rangle & \text{if } \exists r \in Regs.\ \exists v \in Vals. \\ & \quad pr(sc) \in \{\textbf{fence}, \textbf{nop}, \\ & \qquad \textbf{mov-rc}\ r\ v\} \\ \langle(sc, \langle\rangle, \langle(r, ia), (r', ia')\rangle)\rangle & \text{if } \exists ia'' \in Iadds. \\ & \quad pr(sc) = \textbf{jge}\ ia''\ r\ r' \wedge \\ & \quad ia = \mathsf{dep}(sc, r, pr) \wedge \\ & \quad ia' = \mathsf{dep}(sc, r', pr) \\ \langle(sc, \langle(d', t')\rangle, \langle(r', ia')\rangle)\rangle & \text{if } \exists r \in Regs.\ \exists v \in Vals. \\ & \quad \exists d \in Dadds. \\ & \quad pr(sc) \in \{\textbf{mov-rm}\ r\ d\ r', \\ & \qquad \textbf{and-rm}\ r\ d\ r', \\ & \qquad \textbf{shr-rm}\ r\ d\ r'\} \wedge \\ & \quad ia' = \mathsf{dep}(sc, r', pr) \wedge \\ & \quad v = \mathtt{getr}(r', rs, b) \wedge \\ & \quad d' = \mathsf{dadd}(d, v) \wedge \\ & \quad t' = \mathtt{cyc}(d', c), \end{cases}$$

where $\mathsf{dep}(ia, r, pr)$ returns the address of the last instruction executed before ia on which the value of register r depends and

$$\mathtt{cyc}(d, c) = \begin{cases} t_{miss} & c(d) \geq 512 \\ t_{hit} & c(d) < 512. \end{cases}$$

If the dispatch stage of the pipeline is idle, the dispatch results in the empty list (Case 1). If the dispatch stage is processing a **fence**, **nop** or **mov-rc** instruction, the dispatch results in a list containing a triple that consists only of the instruction address and two empty lists, because there are no memory-entries to retrieve and no register dependencies (Case 2). If the instruction to be dispatched is a **jge** instruction, the dispatch results in a list that consists of a triple with an empty memory-access list but a non-empty register-dependence list (Case 3). The register-dependence list contains a pair for each of the two registers that are read by the instructions. The pair contains the respective register and the instruction address of the last instruction on which the value of this register depends. Finally, if the instruction to be dispatched is an instruction that reads both, a register value and a memory value, both the memory-access list and the register-dependence list are non-empty (Case 4). The register-dependence list is created in the same way as for **jge** instructions. The memory-access list is contains a pair that consists of the address d' that the instruction accesses and the number t' of cycles required to retrieve the value from d'. The address d' is determined by adding the base address (provided in the instruction) to the value of the offset register (determined by function `getr`) using the function `dadd`. The number of cycles is determined by the function `cyc`, based on whether the entry is cached or not.

We capture the progress of the ongoing retrieval of memory operands by an update of the remaining access times using the function $\mathtt{uts} : RSStates \to RSStates$, such that

$$\mathtt{uts}(stats) = \begin{cases} stats & \text{if } stats = \langle\rangle \\ rc \bullet \mathtt{uts}(stats') & \text{if } stats = \langle(ia, ml, rl)\rangle \bullet stats' \wedge \\ & \quad rc = (ia, \mathtt{ut}(ml), rl), \end{cases}$$

with

$$\mathtt{ut}(ml) = \begin{cases} ml & \text{if } ml = \langle\rangle \\ (d, t-1) \bullet \mathtt{ut}(ml') & \text{if } ml = \langle(d, t)\rangle \bullet ml' \wedge t > 0 \\ (d, t) \bullet \mathtt{ut}(ml') & \text{if } ml = \langle(d, t)\rangle \bullet ml' \wedge t \leq 0. \end{cases}$$

The function decreases the access time for each entry in the memory-access lists in a list of stations by 1 unless the time has already reached 0, i.e., the memory entry is available already.

Recall that, Definition 7 selects a memory access from the memory-access lists in a reservation-station-configuration $stats$ where the remaining access time is $t_{hit} + 1$. We argued that Definition 7 is well-founded because each instruction accesses at most one memory entry. Note that, for any reasonable choice of access times ($t_{miss} > t_{hit} > 0$) this matches the above definitions of `dodisp` and `uts`. The list of entries to be added to the reservation stations, as defined by `dodisp` contains at most one memory access, which either has an access time below $t_{hit} + 1$ or has access time t_{miss}. Since at most one access is added in each clock cycle and since `uts` decrements the remaining access times in each clock cycle, there can never be two accesses with remaining access time $t_{hit} + 1$.

We capture the removal of the next ready instruction by the *station-purge function* $\mathtt{pus} : RSStates \to RSStates$, such that

$$\mathtt{pus}(stats) = \begin{cases} stats & \text{if } stats = \langle\rangle \\ stats' & \text{if } stats = rc \bullet stats' \\ & \quad \wedge\ \mathsf{addr}(rc) = \mathtt{next}(stats, \langle\rangle) \\ rc \bullet \mathtt{pus}(stats') & \text{if } stats = rc \bullet stats' \\ & \quad \wedge\ \mathsf{addr}(rc) \neq \mathtt{next}(stats, \langle\rangle). \end{cases}$$

This function removes the next ready instruction from a given list of reservation stations. Recall from Definition 8 that the same instruction is moved into the pipeline's execute state. That is, the processing of the instruction is captured faithfully and the instruction does not get lost by the removal from the reservation stations.

The following captures the overall update of reservation stations.

Definition 10. The *reservation-station update* is $\mathtt{c\text{-}upd}_{\mathtt{rs}} : RSStates \times PStates \times Progs_{wf} \times CStates \times RStates \to RSStates$, s.t.

$$\mathtt{c\text{-}upd}_{\mathtt{rs}}(stats, pi, pr, c, rs) = \\ \mathtt{uts}(\mathtt{pus}(stats)) \bullet \mathtt{dodisp}(\mathtt{dstage}(pi), pr, c, rs, \mathtt{buf}(pi)).$$

That is, the executed instruction (if any) is removed from the stations, the newly dispatched instruction is added to the stations, and the memory-access lists in the remaining stations are updated.

Note that, we do not enforce an upper limit on the number of instructions in the reservation stations. In practice, the finite size of such components would limit the amount of out-of-order execution and, hence, the depth of speculative execution. In our model, the depth is limited by the time t_{miss} required to retrieve memory entries. In particular, the next instruction to be executed (selected

by next) is always the first instruction that entered the reservation stations and is ready to be executed.

Reorder Buffer. In each clock cycle, the reorder buffer is updated by removing the entry (if any) that is committed in the current cycle and adding a new entry for the instruction (if any) that is dispatched in the current cycle. We capture the removal of the committed entry by $\mathsf{pub} : BStates \times StConts \to BStates$, s.t.

$$\mathsf{pub}(b, sc) = \begin{cases} b & \text{if } b = \langle\rangle \\ \mathsf{pub}(b') & \text{if } b = \langle bc\rangle \bullet b' \wedge sc = \mathsf{addr}(bc) \\ bc \bullet \mathsf{pub}(b') & \text{if } b = \langle bc\rangle \bullet b' \wedge sc \neq \mathsf{addr}(bc) \end{cases}.$$

The function $\mathsf{pub}(b, sc)$ removes the entry from the buffer b that corresponds to the pipeline contents sc. If applied to a buffer and the contents of the commit stage, the function faithfully captures the removal of the committed register update from the reorder buffer. If the buffer is empty (Case 1) or the commit stage is idle (Case 3), no entry is removed from the buffer.

We capture the computation of the new entry to be added to the buffer for a given pipeline contents by the *register-result function* $\mathtt{result} : StConts \times Progs_{wf} \times BStates \times MStates \times RStates \to BStates$, such that

$$\mathtt{result}(sc, pr, b, m, rs) = \begin{cases} \langle\rangle & \text{if } sc \notin Iadds \vee \\ & (\exists ia \in Iadds.\ \exists r, r' \in Regs. \\ & \quad pr(sc) \in \{\textbf{jge}\ ia\ r\ r', \\ & \qquad \textbf{fence}, \textbf{nop}\}) \\ \langle (sc, r, v)\rangle & \text{if } (pr(sc) = \textbf{mov-rc}\ r\ v) \vee \\ & (\exists d \in Dadds.\ \exists r' \in Regs. \\ & \quad pr(sc) = \textbf{mov-rm}\ r\ d\ r' \wedge \\ & \quad v = m(\mathsf{dadd}(d, \mathtt{getr}(r', rs, b)))) \\ \langle (sc, r, v \&\& v')\rangle & \text{if } \exists d \in Dadds.\ \exists r' \in Regs. \\ & \quad pr(sc) = \textbf{and-rm}\ r\ d\ r' \wedge \\ & \quad v = \mathtt{getr}(r, rs, b) \wedge \\ & \quad v' = m(\mathsf{dadd}(d, \mathtt{getr}(r', rs, b))) \\ \langle (sc, r, v' >> v)\rangle & \text{if } \exists d \in Dadds.\ \exists r' \in Regs. \\ & \quad pr(sc) = \textbf{shr-rm}\ r\ d\ r' \wedge \\ & \quad v = \mathtt{getr}(r, rs, b) \wedge \\ & \quad v' = m(\mathsf{dadd}(d, \mathtt{getr}(r', rs, b))), \end{cases}$$

where $v \&\& v'$ is the bit-wise AND between v and v' and where $v' >> v$ is the logical right-shift of v' by v positions.

If the pipeline contents is not an instruction (i.e., the pipeline stage is empty) or an instruction that does not write to a register, the result to be added to the reorder buffer is just the empty list (Case 1). If the pipeline contents is a **mov-rc** instruction that moves a constant value to a register, the corresponding triple of instruction address, register and value is the result to be added to the reorder buffer (Case 2). In case of a **mov-rm** instruction, the value for the triple to be added to the reorder buffer is determined by adding the base address from the instruction to the value of the offset register and looking up the corresponding value from the given memory configuration (Case 2). In case of an **and-rm** (Case 3) or **shr-rm** (Case 4) instruction, the value from the memory is combined by bit-wise and with the value from the destination register or shifted to the right by the value from the destination register, respectively.

Recall that, Definition 6 selects an entry of the reorder buffer that captures the register update triggered by an instruction ia. We argued that Definition 6 is well-founded because there are no backwards jumps and each instruction writes to at most one register. Note that, this matches the above definition of result. The list of entries to be added to the reorder buffer for the current instruction, as defined by result, always consists of at most one element.

Definition 11. The *reorder-buffer update* is the function $\mathtt{c\text{-}upd}_{\mathsf{buf}} : BStates \times PStates \times Progs_{wf} \times MStates \times RStates \to BStates$, s.t.

$$\mathtt{c\text{-}upd}_{\mathsf{buf}}(b, pi, pr, m, rs) = \mathsf{pub}(b, \mathtt{cstage}(pi)) \bullet \mathtt{result}(\mathtt{dstage}(pi), pr, b, m, rs).$$

That is, the overall update of the reorder buffer combines the removal of the committed entry (if any) with the addition of the entry for the newly dispatched instruction.

Again, we do not enforce an upper limit on the size of the reorder buffer. As explained for the reservation-station update, the speculation depth is limited implicitly by the time t_{miss} after which all memory operands of an instruction are available.

Overall Pipeline Update. We combine the pipeline-stage update, the pipeline flush, the reservation-station update and the reorder-buffer updates to capture the overall effect of one clock cycle during program execution on the instruction pipeline.

Definition 12. The *concrete pipeline update* is the function $\mathtt{c\text{-}upd}_{\mathrm{pipe}} : PStates \times Progs_{wf} \times RStates \times MStates \times CStates \to PStates$, s.t.

$$\mathtt{c\text{-}upd}_{\mathrm{pipe}}(pi, pr, rs, m, c) = \begin{cases} (\mathtt{flush}(pi, pr), \langle\rangle, \langle\rangle) & \text{if } \mathtt{isfl}(pi, pr, rs) \\ (\mathtt{c\text{-}upd}_{\mathtt{st}}(pi, pr), & \text{otherwise.} \\ \quad \mathtt{c\text{-}upd}_{\mathtt{rs}}(\mathtt{stat}(pi), pi, pr, c, rs), & \\ \quad \mathtt{c\text{-}upd}_{\mathsf{buf}}(\mathsf{buf}(pi), pi, pr, m, rs)) & \end{cases}$$

That is, in case of a pipeline flush (Case 1), the pipeline stages are cleared based on flush, and the reservation stations and reorder buffer are emptied. In case of a regular pipeline update (no flush), the stages, reservation stations and reorder buffer are updated component-wise according to their respective update functions.

Putting all together. Since each of the component-update functions takes into account the relevant influence of the other components, we define the overall concrete update function as follows.

Definition 13. The *concrete update* is the function $\mathtt{c\text{-}upd} : States \to States$, such that

$$\begin{aligned} \mathtt{c\text{-}upd}(pr, rs, m, c, pi) = \\ & (\mathtt{c\text{-}upd}_{\mathrm{pr}}(pr), \mathtt{c\text{-}upd}_{\mathrm{reg}}(rs, \mathsf{buf}(pi), \mathtt{cstage}(pi)), \\ & \mathtt{c\text{-}upd}_{\mathrm{mem}}(m), \mathtt{c\text{-}upd}_{\mathrm{cache}}(c, \mathtt{stat}(pi)), \\ & \mathtt{c\text{-}upd}_{\mathrm{pipe}}(pi, pr, rs, m, c)) \end{aligned}$$

That is, the concrete update function updates a configuration component-wise. Overall, the concrete update function models the effect of one clock cycle during the execution of a well-formed pASM program in the environment captured by the concrete configurations. Based on the definitions of the component updates, it

faithfully captures this effect for a static always-not-taken branch-prediction strategy and FIFO cache-replacement policy.

We capture the effect of an entire program execution, starting from initial configuration $state \in States$ by the fixed point of the update function $fix(\texttt{c-upd}(cstate))$.

In the following, we define a program analysis that computes upper bounds on the cache-side-channel leakage of programs with respect to this model of execution.

4 PROGRAM ANALYSIS

As described in Section 2.2, we use a combination of information theory and abstract interpretation to compute quantitative upper bounds on the cache-side-channel leakage of programs in *p*ASM.

We consider an attacker who can observe the entire cache state after the victim code is executed. This abstracts from synchronous access-based cache side channel attacks, which might use different techniques like Prime+Probe or Evict+Time [30] to obtain information about the cache state after the execution.

We quantify the leakage of a program pr executed on initial states from the set $Init$ by the logarithm of the number of possible observations under this attacker model, i.e., by

$$\log|\{c \mid \exists init \in Init.\ \exists rs \in RStates.\ \exists m \in MStates.\ \exists pi \in PStates.\ \texttt{fix}(\texttt{c-upd}(init)) = (pr, rs, m, c, pi)\}|$$

Note that, this bounds the reduction in uncertainty that a one-try attacker has about the set $Init$. That is, $Init$ must be selected in a way that captures the attacker's knowledge faithfully. That is, $Init$ should cover all possible combinations of values for registers or memory entries that are not known to the attacker. This might result in a very large set $Init$ and might make the computation of the leakage bound infeasible based on c-upd.

To make the computation tractable, we view the configurations from $States$ as our concrete domain and the function c-upd as our concrete semantics. We define an abstract domain and abstract semantics to overapproximate the possible concrete executions.

4.1 Abstract Domain

We abstract from register configurations, memory configurations and cache configurations using a set abstraction.

We abstract from concrete configurations component-wise. For registers, memory and cache, we use a set abstraction. Formally, this means that the set of abstract register configurations is the set $RStates^\# = Regs \to \mathcal{P}(Vals)$, the set of abstract memory configurations is the set $MStates^\# = Dadds \to \mathcal{P}(Vals)$, and the set of abstract cache configurations is the set $CStates^\# = Dadds \to \mathcal{P}(\mathbb{N})$.

In abstract pipeline configurations, we abstract from the memory addresses in the reservation stations by sets. That is, $RSStates^\# = RSConts^{\#\star}$ with $RSConts^\# = Iadds \times MLists^\# \times RLists$ and with $MLists^\# = (\mathcal{P}(Dadds) \times \mathbb{N})^\star$. We also abstract from the register values in the reorder buffer by sets. That is, $BStates^\# = BConts^{\#\star}$ with $BConts^\# = Iadds \times Regs \times \mathcal{P}(Vals)$. We do not abstract from the contents of the pipeline stages at this level. Overall, the set of abstract pipeline configurations is the set

$$PStates^\# = StStates \times RSStates^\# \times BStates^\#.$$

Based on the abstract register, memory and cache configurations, we define the overall set of abstract configurations as follows.

Definition 14. The *set of abstract configurations* is the set

$$States^\# = (Progs_{wf} \times PStates^\#) \to (RStates^\# \times MStates^\# \times CStates^\#).$$

That is, an abstract configuration is a mapping from a program and an abstract pipeline configuration to abstract register, memory and cache configurations. Thereby, we preserve relational information between the state of the pipeline and the contents of memory. This allows us to define a more precise abstract semantics. In our evaluation on different target code snippets in the style of the basic Spectre v1 snippet, this precision did not introduce prohibitive analysis complexity (see Section 5).

We use the function names fstage, dstage, estage, cstage, stages, stat, and buf to denote variants of these selector functions that are lifted to abstract pipeline configurations.

Given a set of concrete configurations, we compute the representation of these configurations in the abstract domain with the *abstraction function* $\alpha : \mathcal{P}(States) \to States^\#$, such that

$$\alpha(S) = \left(\lambda(pr, pi^\#).\ \begin{cases} (\alpha_{\mathrm{r}}(rs), \alpha_{\mathrm{m}}(m), \alpha_{\mathrm{c}}(c)) & \text{if } (\exists pi \in PStates.\ pi^\# = \alpha_{\mathrm{pi}}(pi) \wedge (pr, rs, m, c, pi) \in S) \\ ((\lambda r.\{\}), (\lambda d.\{\}), (\lambda d.\{\})) & \text{otherwise} \end{cases} \right)$$

Here, $\alpha_{\mathrm{r}}(rs) = (\lambda r.\ \{rs(r)\})$ abstracts from the register values, $\alpha_{\mathrm{m}}(m) = (\lambda d.\ \{m(d)\})$ abstracts from the memory values, and $\alpha_{\mathrm{c}}(c) = (\lambda d.\ \{c(d)\})$ abstracts from the cache lines.

The abstraction function for the instruction pipeline is

$$\alpha_{\mathrm{pi}}((stags, stats, b)) = (stags, \alpha_{\mathrm{stats}}(stats), \alpha_{\mathrm{buf}}(b)),$$

which is defined based on the reorder-buffer abstraction function

$$\alpha_{\mathrm{buf}}(b) = \begin{cases} \langle\rangle & \text{if } b = \langle\rangle \\ \langle (r, \{v\}) \rangle \bullet \alpha_{\mathrm{buf}}(b') & \text{if } b = \langle (r, v) \rangle \bullet b' \end{cases}$$

and the reservation-station abstraction function

$$\alpha_{\mathrm{stats}}(stats) = \begin{cases} \langle\rangle & \text{if } stats = \langle\rangle \\ \langle (ia, \alpha_{\mathrm{acc}}(ml), rl) \rangle \bullet \alpha_{\mathrm{stats}}(stats') & \text{if } stats = \langle (ia, ml, rl) \rangle \bullet stats' \end{cases}$$

with

$$\alpha_{\mathrm{acc}}(ml) = \begin{cases} \langle\rangle & \text{if } ml = \langle\rangle \\ \langle (\{d\}, n) \rangle \bullet \alpha_{\mathrm{acc}}(ml') & \text{if } ml = \langle (d, n) \rangle \bullet ml'. \end{cases}$$

Essentially, the abstraction functions just propagate the set abstraction downwards in the structure of the configurations.

Analogously, we compute the set of concrete configurations represented by a given abstract configuration using the concretization function $\gamma : States^\# \to \mathcal{P}(States)$, such that

$$\gamma(state^\#) = \{(pr, rs, m, c, pi) \mid \exists pi^\# \in PStates^\#.\ \exists rs^\# \in RStates^\#.\ \exists m^\# \in MStates^\#.\ \exists c^\# \in CStates^\#.\ state^\#(pr, pi^\#) = (rs^\#, m^\#, c^\#) \wedge pi \in \gamma_{\mathrm{pi}}(pi^\#) \wedge rs \in \gamma_{\mathrm{r}}(rs^\#) \wedge m \in \gamma_{\mathrm{m}}(m^\#) \wedge c \in \gamma_{\mathrm{c}}(c^\#)\}$$

Like the abstraction function, the concretization function is based on auxiliary functions that concretize the individual components of the configuration. In particular, $\gamma_{\mathrm{r}}(rs^{\#}) = \{rs \mid \forall r.\ rs(r) \in rs^{\#}(r)\}$ concretizes the register values, $\gamma_{\mathrm{m}}(m^{\#}) = \{m \mid \forall d.\ m(d) \in m^{\#}(d)\}$ concretizes the memory values, and $\gamma_{\mathrm{c}}(c^{\#}) = \{c \mid (\forall d.\ c(d) \in c^{\#}(d)) \wedge (\forall n < 512. \forall d, d' \in Dadds.\ d \neq d' \wedge c(d) = n \Rightarrow c(d') \neq n)\}$ concretizes the cache lines. The concretization of the cache lines takes into account that two different memory blocks cannot be cached in the same cache line simultaneously.

For the instruction pipeline, the concretization function

$$\gamma_{\mathrm{pipe}}((stags, stats^{\#}, b^{\#})) = \{(stags, stats, b) \mid stats \in \gamma_{\mathrm{stats}}(stats^{\#}) \wedge b \in \gamma_{\mathrm{buf}}(b^{\#})\}$$

is defined with respect to the concretization function

$$\gamma_{\mathrm{buf}}(b^{\#}) = \{b \mid |b| = |b^{\#}| \wedge (\forall i < |b|.\ b_i = (r, v) \Rightarrow (\exists VS.\ b_i^{\#} = (r, VS) \wedge v \in VS))\}$$

for the reorder buffer and with respect to

$$\gamma_{\mathrm{stats}}(stats^{\#}) = \{stats \mid |stats| = |stats^{\#}| \wedge (\forall i < |stats|.\ stats_i = (ia, ml, rl) \Rightarrow (\exists ml^{\#}.\ stats_i^{\#} = (ia, ml^{\#}, rl) \wedge ml \in \gamma_{\mathrm{ml}}(ml^{\#})))\}$$

for the reservation stations with the auxiliary function

$$\gamma_{\mathrm{ml}}(ml^{\#}) = \{ml \mid |ml| = |ml^{\#}| \wedge (\forall i < |ml|.\ ml_i = (d, n) \Rightarrow (\exists DS.\ ml_i^{\#} = (DS, n) \wedge d \in DS))\}.$$

The abstraction and concretization functions define the conversion between concrete and abstract states. We define the join across multiple abstract states by $\bigsqcup : \mathcal{P}(States^{\#}) \to States^{\#}$, such that

$$\bigsqcup(S^{\#}) = \left(\lambda(pr, pi^{\#}).\ \bigsqcup_{\mathrm{x}}(\{state^{\#}(pr, pi^{\#}) \mid state^{\#} \in S^{\#}\})\right).$$

Here, $\bigsqcup_{\mathrm{x}}(X^{\#}) = (\bigsqcup_{\mathrm{r}}\{rs^{\#} \mid \exists m^{\#}.\ \exists c^{\#}.\ (rs^{\#}, m^{\#}, c^{\#}) \in X^{\#}\}, \bigsqcup_{\mathrm{m}}\{m^{\#} \mid \exists rs^{\#}.\ \exists c^{\#}.\ (rs^{\#}, m^{\#}, c^{\#}) \in X^{\#}\}, \bigsqcup_{\mathrm{c}}\{c^{\#} \mid \exists rs^{\#}.\ \exists m^{\#}.\ (rs^{\#}, m^{\#}, c^{\#}) \in X^{\#}\})$ is the join across triples of abstract register, memory and cache configurations. It performs a component-wise join using the function $\bigsqcup_{\mathrm{r}}(R^{\#}) = (\lambda r.\ \{v \mid \exists rs^{\#} \in R^{\#}.\ v \in rs^{\#}(r)\})$ for registers, $\bigsqcup_{\mathrm{m}}(M^{\#}) = (\lambda d.\ \{v \mid \exists m^{\#} \in M^{\#}.\ v \in m^{\#}(d)\})$ for memory, and $\bigsqcup_{\mathrm{m}}(C^{\#}) = (\lambda d.\ \{v \mid \exists c^{\#} \in C^{\#}.\ v \in c^{\#}(d)\})$ for cache.

That is, the join function computes the least upper bound of abstract configurations by applying the set union across all values that can occur for a given register, memory entry or cache line in the set $X^{\#}$ under the same abstract pipeline configuration.

4.2 Abstract Semantics

We define the abstract semantics for a clock cycle by an abstract variant of the concrete update function. Again, we first define the abstract update functions for the individual components.

Memory and Registers. As in the concrete case, the abstract update functions for the memory are just the identity functions.

Definition 15. The *abstract instruction-memory update* is the function $\mathtt{a\text{-}upd}_{\mathrm{pr}} : Progs_{wf} \to Progs_{wf}$, such that $\mathtt{a\text{-}upd}_{\mathrm{pr}}(pr) = pr$.

Definition 16. The *abstract data-memory update* is the function $\mathtt{a\text{-}upd}_{\mathrm{mem}} : MStates^{\#} \to MStates^{\#}$, s.t. $\mathtt{a\text{-}upd}_{\mathrm{mem}}(m^{\#}) = m^{\#}$.

Like the concrete register update, the abstract register update depends on the reorder buffer and the instruction to be committed.

Definition 17. The *abstract register update* is the function $\mathtt{a\text{-}upd}_{\mathrm{reg}} : RStates^{\#} \times BStates^{\#} \times StConts \to RStates^{\#}$, such that

$$\mathtt{a\text{-}upd}_{\mathrm{reg}}(rs^{\#}, b^{\#}, ia) = \left(\lambda r.\ \begin{cases} VS & \text{if } \exists i < |b^{\#}|.\ b_i^{\#} = (ia, r, VS) \\ regs^{\#}(r) & \text{otherwise} \end{cases}\right)$$

That is, the abstract register update simply lifts the concrete register update to sets of registers and therefore reliably overapproximates all possible concrete updates.

As in the concrete case, the definition is well-founded because there is at most one entry in the reorder buffer for each instruction. The reason is that each instruction updates at most one register and occurs at most once due to the absence of backwards jumps.

Cache. In an abstract execution, the memory entries that might be accessed during a clock cycle are represented by an abstract value, namely the set of memory entries that might be accessed. We first define how an access to one memory entry would affect the abstract cache state and then lift the definition to the general case.

The function $\mathtt{a\text{-}upd}_{\mathrm{c}} : CStates^{\#} \to Dadds \to CStates^{\#}$ with

$$\mathtt{a\text{-}upd}_{\mathrm{c}}(c^{\#}, acc) = \left(\lambda d.\ \begin{cases} \{n \mid (n \in c^{\#}(d) \wedge n < 512) \vee (n = 0 \wedge 512 \in c^{\#}(d))\} & \text{if } d = acc \\ \{n \mid (n \in c^{\#}(d) \wedge n < 512 \wedge (\exists n' < 512.\ n' \in c^{\#}(acc))) \vee (n - 1 \in c^{\#}(d) \wedge n \leq 512 \wedge 512 \in c^{\#}(acc))\} & \text{otherwise,} \end{cases}\right)$$

returns an abstract cache state that maps each memory entry to updated sets of cache lines, which capture the possible positions of this entry in the cache after an access to *acc*. For the memory entry *acc* itself (Case 1), the set contains the previously possible positions that are inside the cache. This captures the case where the accessed memory entry was already in the cache before the access and the cache remains unchanged. In addition, the set contains position 0 if the entry might have been outside the cache in the previous state. For all other memory entries (Case 2), the set contains the previous positions of these entries if the accessed entry might have already been inside the cache. In addition, it contains the previous positions that are shifted by 1 in case the accessed memory entry might not yet be cached. This captures that the memory entry is added to the cache and all other entries are shifted to make room.

Definition 18. The *abstract cache update* is the function $\mathtt{a\text{-}upd}_{\mathrm{cache}} : CStates^{\#} \times RSStates^{\#} \to CStates^{\#}$, such that

$\mathtt{a\text{-}upd}_{\mathrm{cache}}(c^\#, stats^\#) =$

$$\left(\lambda d'.\ \begin{cases} \{\mathtt{a\text{-}upd}_{\mathrm{c}}(c^\#, d)(d') \mid d \in DS\} & \text{if } \exists ia.\ \exists ml^\#.\exists rl.\\ & \exists i < |stats^\#|.\\ & \exists j < |ml^\#|.\\ & stats^\#_i = (ia, ml^\#, rl) \wedge\\ & ml^\#_j = (DS, t_{hit}+1)\\ c^\#(d') & \text{otherwise} \end{cases}\right)$$

The abstract cache-update function $\mathtt{a\text{-}upd}_{\mathrm{cache}}$ is defined on an abstract cache state and a reservation-station configuration. If there is no instruction whose memory operand has only access time $t_{hit}+1$ remaining, the cache remains unchanged. Otherwise, the new abstract cache state maps each memory entry to the union of all possible new positions that it might have if any of the possible operand addresses is accessed. Thus, the effects of the corresponding concrete cache updates are overapproximated reliably.

Again, the definition is well-founded because there is at most one memory access (represented by an access to a set of memory addresses) with remaining access time $t_{hit}+1$. The reason is that each instruction performs at most one memory access.

Pipeline. As in the concrete semantics, we define the abstract pipeline update based on update functions for the individual components: the pipeline stages, reservation stations and reorder buffer.

Pipeline Stages. We first define an abstract version of `next`. The predicate $\mathtt{a\text{-}ready} : RSConts^\# \times RSStates^\# \to \mathbb{B}$ with
$\mathtt{a\text{-}ready}((ia, ml^\#, rl), stats^\#) =$

$$\begin{aligned}&(\forall i < |ml^\#|.\ \exists DS.\ ml^\#_i = (DS, 0) \wedge\\ &(\forall j < |rl|.\ \exists r \in Regs.\ \exists ia' \in Iaddrs.\\ &\quad rl_j = (r, ia') \wedge (\forall k < |stats^\#|.\ \mathtt{addr}(stats^\#_k) \neq ia'))).\end{aligned}$$

checks for a given instruction from the reservation stations whether the memory operands are available and whether all register dependencies have been resolved. The function
$\mathtt{a\text{-}next}(stats^\#, prevs^\#) =$

$$\begin{cases} \uparrow & \text{if } stats^\# = \langle\rangle\\ \mathtt{addr}(rc^\#) & \text{if } stats^\# = \langle rc^\#\rangle \bullet stats'^\# \wedge\\ & \quad \mathtt{a\text{-}ready}(rc^\#, prevs^\#)\\ \mathtt{a\text{-}next}(stats'^\#, prevs^\# \bullet \langle rc^\#\rangle) & \text{if } stats^\# = \langle rc^\#\rangle \bullet stats'^\# \wedge\\ & \quad \neg\mathtt{a\text{-}ready}(rc^\#, prevs^\#). \end{cases}$$

lifts the concrete function `next` to abstract reservation stations.

To capture the retrieval of abstract register values, we define

$\mathtt{a\text{-}getr}(r, rs^\#, b^\#) =$

$$\begin{cases} rs^\#(r) & \text{if } b^\# = \langle\rangle\\ \mathtt{a\text{-}getr}(r, rs^\#, b'^\#) & \text{if } \exists ia \in Iadds.\ \exists r' \in Regs.\\ & \quad b^\# = b'^\# \bullet \langle(ia, r', VS)\rangle \wedge r' \neq r\\ VS & \text{if } \exists ia \in Iadds.\ b^\# = b'^\# \bullet \langle(ia, r, VS)\rangle \end{cases}$$

This function works like the concrete register-value function, but instead of retrieving an individual values it retrieves the set of values that captures the possible values of the respective register.

We define the abstract pipeline-stage update as follows.

Definition 19. The *abstract pipeline-stage update* is the function $\mathtt{a\text{-}upd}_{\mathtt{st}} : PStates^\# \times Progs_{wf} \to StStates$, such that
$\mathtt{a\text{-}upd}_{\mathtt{st}}(pi^\#, pr) =$

$$\left(\lambda s.\ \begin{cases} \mathtt{iadd}(\mathtt{fstage}(pi^\#), 1) & \text{if } s = fet \wedge\\ & \quad \mathtt{fstage}(pi^\#) \in Iadds \wedge\\ & \quad \neg pr(\mathtt{fstage}(pi^\#)) = \textbf{fence} \wedge\\ & \quad \neg pr(\mathtt{iadd}(\mathtt{fstage}(pi^\#), 1))\uparrow\\ \top & \text{if } s = fet \wedge\\ & \quad (\mathtt{fstage}(pi^\#) = \top \vee\\ & \quad\ (\mathtt{fstage}(pi^\#) \in Iadds \wedge\\ & \quad\quad pr(\mathtt{iadd}(\mathtt{fstage}(pi^\#), 1))\uparrow))\\ \bot & \text{if } (s = fet \wedge\\ & \quad\quad (pr(\mathtt{fstage}(pi^\#)) = \textbf{fence} \vee\\ & \quad\quad (\mathtt{fstage}(pi^\#) = \bot \wedge\\ & \quad\quad (pr(\mathtt{dstage}(pi^\#)) = \textbf{fence} \vee\\ & \quad\quad \exists ia \in Iadds.\ \exists i < |\mathtt{stat}(pi^\#)|.\\ & \quad\quad\quad ia = \mathtt{addr}(\mathtt{stat}(pi^\#)_i) \wedge\\ & \quad\quad\quad pr(ia) = \textbf{fence}))))) \vee\\ & \quad (s = exe \wedge\\ & \quad\quad \mathtt{a\text{-}next}(\mathtt{stat}(pi^\#), \langle\rangle)\uparrow)\\ \mathtt{fstage}(pi^\#) & \text{if } s = dis\\ \mathtt{a\text{-}next}(\mathtt{stat}(pi^\#), \langle\rangle) & \text{if } s = exe \wedge \neg\mathtt{a\text{-}next}(\mathtt{stat}(pi^\#), \langle\rangle)\uparrow\\ \mathtt{estage}(pi^\#) & \text{if } s = com \end{cases}\right).$$

Since the contents of each stage is a concrete value and since the next ready instruction is uniquely defined, the function is a straightforward lifting of the concrete update to the abstract case.

We lift the concrete flush predicate to the abstract case as follows.

$\mathtt{a\text{-}isfl}(pi^\#, pr, rs^\#) =$

$$\begin{cases} \{\text{true}\} & \text{if } \exists ia \in Iadds.\ \exists r, r' \in Regs.\\ & \quad pr(\mathtt{estage}(pi^\#)) = \textbf{jge}\ ia\ r\ r' \wedge\\ & \quad \forall v \in \mathtt{a\text{-}getr}(r, rs^\#, \mathtt{buf}(pi^\#)).\\ & \quad\ \forall v' \in \mathtt{a\text{-}getr}(r', rs^\#, \mathtt{buf}(pi^\#)).\\ & \quad\quad v \geq v'\\ \{\text{false}\} & \text{if } (\neg(\exists ia \in Iadds.\ \exists r, r' \in Regs.\\ & \quad pr(\mathtt{estage}(pi^\#)) = \textbf{jge}\ ia\ r\ r')) \vee\\ & \quad (\exists ia \in Iadds.\ \exists r, r' \in Regs.\\ & \quad pr(\mathtt{estage}(pi^\#)) = \textbf{jge}\ ia\ r\ r' \wedge\\ & \quad \forall v \in \mathtt{a\text{-}getr}(r, rs^\#, \mathtt{buf}(pi^\#)).\\ & \quad\ \forall v' \in \mathtt{a\text{-}getr}(r', rs^\#, \mathtt{buf}(pi^\#)).\\ & \quad\quad v < v')\\ \{\text{true}, \text{false}\} & \text{otherwise} \end{cases}$$

If all register values captured by the abstract register configuration are such that a pipeline flush would occur, the lifted predicate returns the singleton set {true}. Conversely, if there is no possibility for a pipeline flush, the predicate returns the singleton set {false}. If both cases are possible, the predicate returns {true, false}. That is, the lifted predicate reliably overapproximates the possibilities with respect to the occurrence of a pipeline flush.

Definition 20. The *abstract pipeline flush* is the function $\texttt{a-flush} : PStates^{\#} \times Progs_{wf} \to StStates$, such that

$$\texttt{a-flush}(pi^{\#}, pr) = \left(\lambda s. \begin{cases} ia & \text{if } s = fet \wedge \exists r, r' \in Regs. \\ & \quad pr(\texttt{estage}(pi^{\#})) = \textbf{jge}\ ia\ r\ r' \\ \top & \text{otherwise} \end{cases} \right).$$

The abstract pipeline flush is again a straightforward lifting of the concrete variant and, thus, a reliable overapproximation of the effects of a pipeline flush in concrete executions.

Reservation Stations. We abstract from the instruction dispatch, the update of memory access times and the instruction purge to capture to obtain an abstract update for the reservation stations.

We define the *abstract instruction dispatch* by function $\texttt{a-dodisp} : StConts \times Progs_{wf} \times CStates^{\#} \times RStates^{\#} \times BStates^{\#} \to \mathcal{P}(RSStates^{\#})$ with

$$\texttt{a-dodisp}(sc, pr, c^{\#}, rs^{\#}, b^{\#}) = \begin{cases} \{\langle\rangle\} & \text{if } sc \notin Iadds \\ \{\langle (sc, \langle\rangle, \langle\rangle)\rangle\} & \text{if } \exists r \in Regs.\ \exists v \in Vals. \\ & \quad pr(sc) \in \{\textbf{fence}, \textbf{nop}, \\ & \qquad \textbf{mov-rc}\ r\ v\} \\ \{\langle (sc, \langle\rangle, \langle (r, d), (r', d')\rangle)\rangle\} & \text{if } \exists ia \in Iadds. \\ & \quad pr(sc) = \textbf{jge}\ ia\ r\ r' \wedge \\ & \quad d = \texttt{dep}(sc, r, pr) \wedge \\ & \quad d' = \texttt{dep}(sc, r', pr) \\ \{S_1, S_2\} & \text{if } \exists r \in Regs.\ \exists d \in Dadds. \\ & \quad pr(sc) \in \{\textbf{mov-rm}\ r\ d\ r', \\ & \qquad \textbf{and-rm}\ r\ d\ r', \\ & \qquad \textbf{shr-rm}\ r\ d\ r'\} \wedge \\ & \quad ia = \texttt{dep}(sc, r', pr) \wedge \\ & \quad S_1 = \langle (sc, \langle (M_1, t_{hit})\rangle, \langle (r', ia)\rangle)\rangle \wedge \\ & \quad S_2 = \langle (sc, \langle (M_2, t_{miss})\rangle, \langle (r', ia)\rangle)\rangle \wedge \\ & \quad M_1 = \{d' \mid \exists v \in Vals. \\ & \qquad v \in \texttt{a-getr}(r', rs^{\#}, b^{\#}) \wedge \\ & \qquad d' = \texttt{dadd}(d, v) \wedge \\ & \qquad t_{hit} \in \texttt{a-cyc}(d', c^{\#})\} \wedge \\ & \quad M_2 = \{d' \mid \exists v \in Vals. \\ & \qquad v \in \texttt{a-getr}(r', rs^{\#}, b^{\#}) \wedge \\ & \qquad d' = \texttt{dadd}(d, v) \wedge \\ & \qquad t_{miss} \in \texttt{a-cyc}(d', c^{\#})\}, \end{cases}$$

where

$$\texttt{a-cyc}(da, c^{\#}) = \begin{cases} \{t_{miss}\} & \text{if } \forall a \in c^{\#}(da).\ a \geq 512 \\ \{t_{hit}\} & \text{if } \forall a \in c^{\#}(da).\ a < 512 \\ \{t_{miss}, t_{hit}\} & \text{otherwise.} \end{cases}$$

Given an abstract cache configuration, a-cyc determines the set of cycle numbers that might be needed to retrieve a memory entry. If the memory entry is cached in all concrete cache configurations captured by the abstract cache configuration, the possible cycle numbers are the singleton set $\{t_{hit}\}$. Analogously, if the memory entry is not cached in any concrete cache configuration captured by the abstract configuration, the possible cycle numbers are the singleton set $\{t_{miss}\}$. If both scenarios, cached and uncached, are possible, the possible cycle numbers are the set $\{t_{miss}, t_{hit}\}$.

The abstract instruction dispatch dispatches those instructions that do not access memory in the same way as the concrete instruction dispatch (Cases 1–3), lifted to sets of abstract reservation-station configurations. For instructions that do access memory, the instruction dispatch generates a set of two abstract reservation-station configurations (Case 4): One configuration that captures the accesses to one of those possible memory addresses that might be cached, and one configuration that captures the access to one of those possible memory addresses that might not be cached. By considering both configurations, it is ensured that no possible attacker observation is overlooked. By considering the configurations separately, the analysis gets sufficiently precise to determine the next ready instruction reliably with a-next.

We lift the concrete update of access times to the abstract case by the function $\texttt{a-uts} : RSStates^{\#} \to RSStates^{\#}$, such that

$$\texttt{a-uts}(stats^{\#}) = \begin{cases} stats^{\#} & \text{if } stats^{\#} = \langle\rangle \\ (ia, ml'^{\#}, rl) \bullet \texttt{a-uts}(stats'^{\#}) & \text{if } stats^{\#} = \langle (ia, ml^{\#}, rl)\rangle \bullet \\ & \quad stats'^{\#} \wedge ml'^{\#} = \texttt{a-ut}(ml^{\#}), \end{cases}$$

where

$$\texttt{a-ut}(ml^{\#}) = \begin{cases} ml^{\#} & \text{if } ml^{\#} = \langle\rangle \\ (DS, n-1) \bullet \texttt{a-ut}(ml'^{\#}) & \text{if } ml^{\#} = \langle (DS, n)\rangle \bullet ml'^{\#} \wedge n > 0 \\ (DS, n) \bullet \texttt{a-ut}(ml'^{\#}) & \text{if } ml^{\#} = \langle (DS, n)\rangle \bullet ml'^{\#} \wedge n \leq 0. \end{cases}$$

Recall that, Definition 18 selects a memory access from the memory-access lists in a reservation-station-configuration *stats* where the remaining access time is $t_{hit} + 1$. We argued that Definition 18 is well-founded because each instruction makes at most one memory access. As in the concrete case, this matches the definitions of a-dodisp and a-uts if $t_{miss} > t_{hit} > 0$.

Analogously, we lift the concrete station-purge function to the function $\texttt{a-pus} : RSStates^{\#} \to RSStates^{\#}$, such that

$$\texttt{a-pus}(stats^{\#}) = \begin{cases} stats^{\#} & \text{if } stats^{\#} = \langle\rangle \\ stats'^{\#} & \text{if } stats^{\#} = rc^{\#} \bullet stats'^{\#} \wedge \\ & \quad \texttt{addr}(rc^{\#}) = \texttt{a-next}(stats^{\#}, \langle\rangle)\ . \\ rc^{\#} \bullet \texttt{a-pus}(stats'^{\#}) & \text{if } stats^{\#} = rc^{\#} \bullet stats'^{\#} \wedge \\ & \quad \texttt{addr}(rc^{\#}) \neq \texttt{a-next}(stats^{\#}, \langle\rangle) \end{cases}$$

Like in the concrete case, the function removes the first instruction from the reservation stations for which the memory operands are available and the register dependencies have been resolved.

We define the abstract reservation-station update as follows.

Definition 21. The *abstract reservation-station update* is $\texttt{a-upd}_{\texttt{rs}} : RSStates^{\#} \times PStates^{\#} \times Progs_{wf} \times CStates^{\#} \times RStates^{\#} \to \mathcal{P}(RSStates^{\#})$, such that

$$\texttt{a-upd}_{\texttt{rs}}(stats^{\#}, pi^{\#}, pr, c^{\#}, rs^{\#}) = \{stats''^{\#} \mid \exists stats'^{\#} \in \texttt{a-dodisp}(\texttt{dstage}(pi^{\#}), pr, c^{\#}, rs^{\#}, \texttt{buf}(pi^{\#})).\ stats''^{\#} = \texttt{a-uts}(\texttt{a-pus}(stats^{\#})) \bullet stats'^{\#}\}.$$

Again, the function returns a set of possible updated abstract reservation-station configurations. Thereby it accounts for the fact that there might be a memory access for which it is unclear whether the target address is cached or not cached. The set of resulting updated abstract reservation-station configurations, thus, reliably overapproximates the possible updates across the concrete executions that the abstract execution represents.

Reorder Buffer. We abstract from the buffer-purge function by the function $\texttt{a-pub} : BStates^{\#} \times StConts \to BStates^{\#}$, such that

$$\texttt{a-pub}(b^{\#}, sc) = \begin{cases} b^{\#} & \text{if } b^{\#} = \langle\rangle \\ \texttt{a-pub}(b'^{\#}) & \text{if } b^{\#} = \langle bc^{\#}\rangle \bullet b'^{\#} \wedge \\ & \quad sc = \texttt{addr}(bc^{\#}) \\ bc^{\#} \bullet \texttt{a-pub}(b'^{\#}) & \text{if } b^{\#} = \langle bc^{\#}\rangle \bullet b'^{\#} \wedge \\ & \quad sc \neq \texttt{addr}(bc^{\#}). \end{cases}$$

As for the abstract station-purge function, the abstract buffer-purge function simply lifts the corresponding concrete function.

To compute the effects of the executed instruction on the buffer, we compute the resulting abstract register updates with $\texttt{a-result} : StConts \times Progs_{wf} \times BStates^{\#} \times MStates^{\#} \times RStates^{\#} \to BStates^{\#}$, where
$\texttt{result}(sc, pr, b^{\#}, m^{\#}, rs^{\#}) =$

$$\begin{cases} \langle\rangle & \text{if } sc \notin Iadds \vee \\ & \quad (\exists ia \in Iadds.\ \exists r, r' \in Regs. \\ & \qquad pr(sc) \in \{\textbf{jge}\ ia\ r\ r', \\ & \qquad\qquad \textbf{fence}, \textbf{nop}\}) \\ \langle (sc, r, \{v\})\rangle & \text{if } pr(sc) = \textbf{mov-rc}\ r\ v \\ \langle (sc, r, X)\rangle & \text{if } \exists d \in Dadds.\ \exists r' \in Regs. \\ & \quad pr(sc) = \textbf{mov-rm}\ r\ d\ r' \wedge \\ & \quad X = \{x \mid \exists v \in \texttt{a-getr}(r', rs^{\#}, b^{\#}). \\ & \qquad x \in m^{\#}(\texttt{dadd}(da, v))\} \\ \langle (sc, r, Y)\rangle & \text{if } \exists d \in Dadds.\ \exists r' \in Regs. \\ & \quad pr(sc) = \textbf{and-rm}\ r\ d\ r' \wedge \\ & \quad Y = \{y_1 \&\& y_2 \mid \\ & \qquad \exists v \in \texttt{a-getr}(r', rs^{\#}, b^{\#}). \\ & \qquad\quad y_1 \in \texttt{a-getr}(r, rs^{\#}, b^{\#}) \wedge \\ & \qquad\quad y_2 \in m^{\#}(\texttt{dadd}(d, v))\} \\ \langle (sc, r, Z)\rangle & \text{if } \exists d \in Dadds.\ \exists r' \in Regs. \\ & \quad pr(sc) = \textbf{shr-rm}\ r\ d\ r' \wedge \\ & \quad Z = \{z_2 >> z_1 \mid \\ & \qquad \exists v \in \texttt{a-getr}(r', rs^{\#}, b^{\#}). \\ & \qquad\quad z_1 \in \texttt{a-getr}(r, rs^{\#}, b^{\#}) \wedge \\ & \qquad\quad z_2 \in mem(\texttt{dadd}(d, v))\}. \end{cases}$$

The function `a-result` returns the empty list in case of an instruction that does not write to a register (Case 1). In case of a **mov-rc** instruction, it returns the list that contains a register update to a singleton set, where the set contains only the constant value from the instruction (Case 2). In case of an instruction that accesses the memory based on an offset register (Case 3–5), the function first retrieves the set of possible values of the offset register. It then computes the set of possible register updates for the destination register, depending on the type of instruction. For a **mov-rm** instruction, it is the union across the sets of values from the abstract memory for each possible address consisting of the base address and one of the possible offsets. For **and-rm** or **shr-rm** instructions, the resulting values are combined with all possible values of the destination register by bit-wise AND or logical shift, respectively. Overall, the register update in the list overapproximates all possible updated values for the target register.

Recall that, Definition 17 selects an entry of the reorder buffer that captures the register update triggered by an instruction *ia*. We argued that Definition 17 is well-founded because there are no backwards jumps and each instruction writes to at most one register. As in the concrete case, this matches the definition of `a-result`.

Definition 22. The *abstract reorder-buffer update* is the function $\texttt{a-upd}_{\texttt{buf}} : BStates^{\#} \times PStates^{\#} \times Progs_{wf} \times MStates^{\#} \times RStates^{\#} \to BStates^{\#}$, such that

$$\begin{aligned} &\texttt{a-upd}_{\texttt{buf}}(b^{\#}, pi^{\#}, pr, m^{\#}, rs^{\#}) = \\ &\qquad \texttt{a-pub}(b^{\#}, \texttt{cstage}(pi^{\#})) \bullet \\ &\qquad\quad \texttt{a-result}(\texttt{dstage}(pi^{\#}), pr, b^{\#}, m^{\#}, rs^{\#}). \end{aligned}$$

The abstract reorder-buffer update combines the abstract register-result function and the abstract buffer-purge function to remove the committed abstract register update (if any) from the buffer and add an abstract register update for the next dispatched instruction.

Overall Pipeline Update. The abstract updates of all pipeline components in scenarios with or without pipeline flushes are combined in the abstract pipeline update as follows.

Definition 23. The *abstract pipeline update* is $\texttt{a-upd}_{\texttt{pipe}} : PStates^{\#} \times Progs_{wf} \times RStates^{\#} \times MStates^{\#} \times CStates^{\#} \to \mathcal{P}(PStates^{\#})$, s.t.

$$\begin{aligned} &\texttt{a-upd}_{\texttt{pipe}}(pi^{\#}, pr, rs^{\#}, m^{\#}, c^{\#}) = \\ &\qquad \{(stags, \langle\rangle, \langle\rangle) \mid stags \in \texttt{a-flush}(pi^{\#}, pr) \wedge \\ &\qquad\quad \text{true} \in \texttt{a-isfl}(pi^{\#}, pr, rs^{\#})\} \\ &\qquad \cup \\ &\qquad \{(stags, stats'^{\#}, b'^{\#}) \mid stags = \texttt{a-upd}_{\texttt{st}}(pi^{\#}, pr) \wedge \\ &\qquad\quad stats'^{\#} \in \texttt{a-upd}_{\texttt{rs}}(\texttt{stat}(pi^{\#}), pi^{\#}, pr, c^{\#}, rs^{\#}) \wedge \\ &\qquad\quad b'^{\#} = \texttt{a-upd}_{\texttt{buf}}(\texttt{buf}(pi^{\#}), pi^{\#}, pr, m^{\#}, rs^{\#}) \wedge \\ &\qquad\quad \text{false} \in \texttt{a-isfl}(pi^{\#}, pr, rs^{\#})\} \end{aligned}$$

The abstract pipeline update is the union of two sets. The first set contains the updated abstract pipeline configuration in case of a pipeline flush or is empty in case no pipeline flush is possible. The second set contains the possible updated abstract pipeline configurations in case no pipeline flush happens. It contains more than one configuration in case there is more than one possible updated abstract reservation-station configuration, i.e., if there is more than one possibility for the clock cycles required to retrieve a memory entry accessed by the instruction that is currently dispatched.

Based on the definitions of the abstract component updates and the treatment of pipeline flushes, the abstract pipeline update reliably overapproximates the effects that one clock cycle during

program execution might have on any of the concrete pipeline configurations represented by the abstract pipeline configuration.

Putting all together. Based on the abstract register, cache and pipeline updates, we define the overall abstract semantics. Since abstract configurations are functions that map the program and abstract pipeline configuration to an abstract configuration for the remaining components, we first define a function that specifies the abstract configuration resulting from the update with respect to one program and abstract pipeline configuration. Afterwards, we define the lifting to the general case.

The *individual abstract update function* is $\mathtt{a\text{-}upd}_{\text{one}} : Progs_{wf} \times PStates^{\#} \times RStates^{\#} \times MStates^{\#} \times CStates^{\#} \to States^{\#}$, such that $\mathtt{a\text{-}upd}_{\text{one}}(pr, pi^{\#}, rs^{\#}, m^{\#}, c^{\#}) = (\lambda(pr', pi'^{\#})$.

$$\begin{cases} (rs'^{\#}, m'^{\#}, c'^{\#}) & \text{if } pr' = \mathtt{a\text{-}upd}_{\text{pr}}(pr) \wedge \\ & \quad pi'^{\#} \in \mathtt{a\text{-}upd}_{\text{pipe}}(pi^{\#}, pr, rs^{\#}, \\ & \qquad m^{\#}, c^{\#}) \wedge \\ & \quad rs'^{\#} = \mathtt{a\text{-}upd}_{\text{reg}}(rs^{\#}, \mathtt{buf}(pi^{\#}), \\ & \qquad \mathtt{cstage}(pi^{\#})) \wedge \\ & \quad m'^{\#} = \mathtt{a\text{-}upd}_{\text{mem}}(m^{\#}) \wedge \\ & \quad c'^{\#} = \mathtt{a\text{-}upd}_{\text{cache}}(c^{\#}, \mathtt{stat}(pi^{\#})) \\ ((\lambda r.\{\}), (\lambda d.\{\}), (\lambda d.\{\})) & \text{otherwise}). \end{cases}$$

The individual abstract update function combines the component-wise abstract cache update, abstract register update and abstract pipeline update for one abstract pipeline configuration.

We lift the abstract update to the general case as follows.

Definition 24. The *abstract update* is $\mathtt{a\text{-}upd} : States^{\#} \to States^{\#}$, such that

$$\mathtt{a\text{-}upd}(state^{\#}) = \bigsqcup \{\mathtt{a\text{-}upd}_{\text{one}}(pr, pi^{\#}, state^{\#}(pr, pi^{\#})) \mid (pr, pi^{\#}) \in Progs_{wf} \times PStates^{\#}\}$$

That is, the overall abstract semantics consists of the join across the individual updates for each abstract pipeline configuration. Based on the definition of the join operator, which computes the least upper bound across a set of abstract configurations, and based on the definitions of the abstract component updates, the abstract update function reliably overapproximates the possible effects of a clock cycle during program execution in the abstract domain.

For a given program $pr \in Progs_{wf}$ and set $Init \in \mathcal{P}(States)$ of initial states (e.g., differing in the value of a secret memory entry), we compute the upper bound on the leakage to the attacker by

$$\log |\{c \mid \exists rs \in RStates.\ \exists m \in MStates.\ \exists pi \in PStates.\ (pr, rs, m, c, pi) \in \gamma(\mathtt{fix}(\mathtt{a\text{-}upd}(\alpha(Init))))\}|.$$

That is, we abstract from the set of initial states, compute the fixed point of the abstract semantics, and count the number of possible cache configurations in the set of concrete final states that are represented by the final abstract state.

In the following, we evaluate our analysis at the different variants of Spectre snippets from Example 3.1 and Example 3.2.

5 EVALUATION

We evaluate our analysis from Section 4 at different variants of Spectre snippets. We instantiate t_{hit} with 1 and t_{miss} with 4.

5.1 Analysis of p-mask

Recall the program p-mask from Example 3.2, which retrieves a memory entry based on the attacker-controlled register *eax*, masks the value to the four least-significant bits and then encodes these into the cache by using them to index an access to a public array.

We first define the set of possible initial configurations and then apply our program analysis to compute an upper bound on the cache-side-channel leakage of the program p-mask. Throughout our evaluation, we use decimal numbers to represent the 32-bit values from the set $Vals$ for better readability.

Let pr_m = p-mask. The initial pipeline configuration is

$$pi_0 = \left(\left(\lambda s.\ \begin{cases} i1 & \text{if } s = fet \\ \top & \text{otherwise} \end{cases}\right), \langle\rangle, \langle\rangle\right),$$

i.e., the pipeline is empty except for the fetch stage that begins with the initial instruction at address $i1$.

Let the set of possible initial memory configurations be

$$M_0 = \left\{ m \mid \exists v \in Vals.\ m = \left(\lambda da.\ \begin{cases} 3 & \text{if } d = d0 \\ v & \text{if } d = d6 \\ 0 & \text{otherwise} \end{cases}\right) \right\}.$$

Recall that $d1$ is the base address of an array $a1$ that spans 3 memory entries, $d6$ is the base address of a second array $a2$, and $d0$ is the data address at which the size of the first array is stored. We consider the case where $a1$ and $a2$ are filled with zeroes and a secret is stored in $d4$ (which might be any 32-bit value in $Vals$).

We consider a scenario in which the attacker targets the value of this secret. Let the initial register configuration be

$$rs_0 = \left(\lambda r.\ \begin{cases} 5 & r = eax \\ 0 & \text{otherwise} \end{cases}\right),$$

i.e., the attacker sets the parameter register *eax* to the value 5.[1]

Finally, let the initial cache configuration be

$$c_0 = \left(\lambda d.\ \begin{cases} 0 & \text{if } d = d6 \\ 512 & \text{otherwise} \end{cases}\right).$$

That is, we assume that the memory entry at and $d4$ is cached and all other entries are not cached.

Overall, the set of initial configurations for program p-mask is

$$Init = \{(pr_m, rs_0, m_0, c_0, pi_0) \mid m_0 \in M_0\}.$$

This captures a typical scenario for Spectre-v1-like attacks, in which the attacker calls the target code snippet with a parameter of his choice to extract secret information from the memory.

[1]Note that, one could consider additional initial register configurations to obtain leakage bounds for more general attack scenarios. For simplicity, we restrict ourselves to one attack scenario in the evaluation.

The first step of our program analysis is to compute the abstract initial configuration $Init^\# = \alpha(Init)$. With this step, we obtain

$$Init^\# = \left(\lambda(pr, pi^\#).\ \begin{cases} (rs_0{}^\#, m_0{}^\#, c_0{}^\#) & \text{if } pr = pr_m \wedge \\ & \quad pi^\# = pi_0{}^\# \\ ((\lambda r.\{\}), (\lambda d.\{\}), (\lambda d.\{\})) & \text{otherwise} \end{cases}\right),$$

where the initial abstract pipeline is

$$pi_0{}^\# = \left(\left(\lambda s.\ \begin{cases} i1 & \text{if } s = fet \\ \top & \text{otherwise} \end{cases}\right), \langle\rangle, \langle\rangle\right),$$

the initial abstract memory configuration is

$$m_0{}^\# = \left(\lambda d.\ \begin{cases} \{3\} & \text{if } d = d0 \\ Vals & \text{if } d = d4 \\ \{0\} & \text{otherwise} \end{cases}\right),$$

and the initial abstract register and cache configurations are

$$rs_0{}^\# = \left(\lambda r.\ \begin{cases} \{3\} & \text{if } r = eax \\ \{0\} & \text{otherwise} \end{cases}\right), c_0{}^\# = \left(\lambda d.\ \begin{cases} \{0\} & \text{if } d = d4 \\ \{512\} & \text{otherwise} \end{cases}\right).$$

The next step of our program analysis is to compute the fixed point $fix(\mathsf{a\text{-}upd}(Init^\#))$ of the abstract update function.

The fixed point of the abstract update is reached after 17 clock cycles (see Appendix A for the detailed computation) at

$$\mathsf{a\text{-}upd}^{17}(Init^\#) = \mathsf{a\text{-}upd}^{16}(Init^\#) = \left(\lambda(pr, pi^\#).\ \begin{cases} (rs_{11}{}^\#, m_0{}^\#, c_{11}{}^\#) & \text{if } pr = pr_m \wedge \\ & \quad pi^\# = pi_{16}{}^\# \\ ((\lambda r.\{\}), (\lambda d.\{\}), (\lambda d.\{\})) & \text{otherwise} \end{cases}\right),$$

where

- $pi_{16}{}^\# = \left(\left(\lambda s.\ \begin{cases} \top & \text{if } s = fet \\ \top & \text{if } s = dis \\ \top & \text{if } s = exe \\ \top & \text{if } s = com \end{cases}\right), \langle\rangle, \langle\rangle\right),$
- $rs_{11}{}^\# = \left(\lambda r.\ \begin{cases} \{3\} & \text{if } r \in \{eax, ebx\} \\ \{v \in Vals \mid 0 \le v \le 15\} & \text{if } r = ecx \\ \{0\} & \text{otherwise} \end{cases}\right),$
- $c_{11}{}^\# = \left(\lambda d.\ \begin{cases} \{1\} & \text{if } d = d0 \\ \{2\} & \text{if } d = d4 \\ \{0, 514\} & \text{if } d \in \{\mathsf{dadd}(d6, v) \mid 0 \le v \le 15\} \\ \{514\} & \text{otherwise} \end{cases}\right).$

The final step of our program analysis is to compute the leakage bound based on the number of different cache configurations in the concretization of $\mathsf{a\text{-}upd}^{16}(Init^\#)$. For this step, we obtain

$$\log|\{c \mid \exists pi \in PStates.\ \exists rs \in RStates.\ \exists m \in MStates.\ (pr, rs, m, c, pi) \in \gamma(\mathsf{a\text{-}upd}^{17}(Init^\#))\}| = \log(16) = 4.$$

That is, the program `p-mask` leaks at most 4bit to the attacker.

Note that, this bound is precise because `p-mask` uses the last four bits of the secret memory entry as the index for a memory access. Thereby, these four bits are encoded into the cache and can be learned by an attacker who observes the final cache configuration.

In the following, we will compare the leakage bound for `p-mask` to the leakage bounds for the other variants of Spectre snippets.

5.2 Analysis of `p-shift`

We define the initial configurations for program `p-shift` analogously to the initial configurations for `p-mask`. Let pr_s = `p-shift`. Then the set of initial configurations for program `p-shift` is

$$Init_s = \{(pr_s, pi_0, rs_0, m_0, c_0) \mid m_0 \in M_0\}.$$

The fixed point of the abstract update function is reached after 17 clock cycles (see Appendix A for the detailed computation) at

$$\mathsf{a\text{-}upd}^{17}(\alpha(Init_s)) = \mathsf{a\text{-}upd}^{16}(\alpha(Init_s)) = \left(\lambda(pr, pi^\#).\ \begin{cases} (rs_{11}{}^{\#\prime}, m_0{}^\#, c_{11}{}^{\#\prime}) & \text{if } pr = pr_s \wedge \\ & \quad pi^\# = pi_{16}{}^\# \\ ((\lambda r.\{\}), (\lambda d.\{\}), (\lambda d.\{\})) & \text{otherwise} \end{cases}\right),$$

where

- $rs_{11}{}^{\#\prime} = \left(\lambda r.\ \begin{cases} \{3\} & \text{if } r \in \{eax, ebx\} \\ \{v \in Vals \mid 0 \le v \le 131071\} & \text{if } r = ecx \\ \{0\} & \text{otherwise} \end{cases}\right),$
- $c_{11}{}^{\#\prime} = \left(\lambda d.\ \begin{cases} \{1\} & \text{if } d = d0 \\ \{2\} & \text{if } d = d4 \\ \{0, 514\} & \text{if } d \in \{\mathsf{dadd}(d6, v) \mid 0 \le v \le 131071\} \\ \{514\} & \text{otherwise} \end{cases}\right).$

Based on this fixed point, we obtain the leakage bound

$$\log|\{c \mid \exists rs \in RStates.\ \exists m \in MStates.\ \exists pi \in PStates.\ (pr, rs, m, c, pi) \in \gamma(\mathsf{a\text{-}upd}^{17}(\alpha(Init_s)))\}| = \log(131072) = 17.$$

That is, the program `p-shift` leaks at most 17bit to the attacker.

Again, the leakage bound is precise, because after shifting out the 15 least-significant bits of the secret value retrieved from the memory, the program `p-shift` uses the remaining 17 bits as the index for an array access. Thereby, it encodes these 17 bits into the cache configuration, which can be observed by the attacker.

5.3 Analysis of `p-simple`

Finally, we define the initial configurations for program `p-simple` analogously to the initial configurations for `p-mask` and `p-shift`.

Let pr_v = `p-simple`. Then the set of initial configurations is

$$Init_v = \{(pr_s, pi_0, rs_0, m_0, c_0) \mid m_0 \in Mem_0\}.$$

The fixed point of the update is reached after 17 clock cycles at

$$\mathsf{a\text{-}upd}^{17}(\alpha(Init_v)) = \mathsf{a\text{-}upd}^{16}(\alpha(Init_v)) = \left(\lambda(pr, pi^\#).\ \begin{cases} (rs_{11}{}^{\#\prime\prime}, m_0{}^\#, c_{11}{}^{\#\prime\prime}) & \text{if } pr = pr_s \wedge \\ & \quad pi^\# = pi_{16}{}^\# \\ ((\lambda r.\{\}), (\lambda d.\{\}), (\lambda d.\{\})) & \text{otherwise} \end{cases}\right),$$

where

- $rs_{11}{}^{\#''} = \left(\lambda r. \begin{cases} \{3\} & \text{if } r \in \{eax, ebx\} \\ \{v \in Vals\} & \text{if } r = ecx \\ \{0\} & \text{otherwise} \end{cases}\right)$
- $c_{11}{}^{\#''} = \left(\lambda d. \begin{cases} \{1\} & \text{if } d = d0 \\ \{2\} & \text{if } d = d4 \\ \{0, 514\} & \text{if } d \in \{\mathsf{dadd}(d6, v) \mid \\ & \quad 0 \leq v \leq 4294967295\} \\ \{514\} & \text{otherwise} \end{cases}\right)$.

Based on this fixed point, we obtain the following leakage bound for `p-simple` with respect to the initial configurations $Init_v$:

$$\log|\{c \mid \exists rs \in RStates.\ \exists m \in MStates.\ \exists pi \in PStates.\ (pr, rs, m, c, pi) \in \gamma(\mathtt{a\text{-}upd}^{17}(\alpha(Init_v)))\}| = \log(4294967296) = 32$$

Based on this bound, `p-simple` leaks at most 32bit to the attacker. That is, it might leak the entire secret memory entry.

5.4 Comparison Across Leakage Bounds

Together with the leakage bounds for `p-mask` and `p-shift`, the bound for `p-simple` allows us to compare the three snippets in terms of their security with respect to our attacker model for cache-side-channel attacks using instruction pipelining with branch prediction and out-of-order execution.

Table 1 summarizes the leakage bounds. While `p-simple` might leak all 32bit of the secret value, we obtain much lower leakage bounds for the snippets `p-shift` and `p-mask`, which might leak at most 17bit and 4bit, respectively.

These bound suggest that for `p-mask` we can rely on the highest degree of security. Indeed, since `p-mask` masks out most of the secret information before the cache leakage, it is the most secure among the three snippets that we have considered. Similarly, the level of security for `p-shift` lies between `p-mask` and `p-simple`.

If snippets of theses types would occur in the same program, we could use leakage bounds like these to guide mitigation efforts, giving the snippets of type `p-simple` the highest priority. Previously, such a quantitative comparison was not possible.

In particular if a program contains many candidate code snippets or if these snippets are too long to inspect manually, a reliable program analysis is key to obtaining a meaningful quantitative assessment of the program. While existing qualitative approaches could be used to identify the candidate snippets of a program that need to be investigated, our quantitative analysis could rank the identified snippets in terms of leakage bounds.

Snippet	Leakage Bound
`p-mask`	4bit
`p-shift`	17bit
`p-simple`	32bit

Table 1: Leakage Bounds across the Snippet Variants

6 RELATED WORK

In the following, we provide an overview of existing work on cache-side-channel analysis and analysis with respect to transient-execution vulnerabilities.

6.1 Cache Side Channels

Cache side channels were first described by Page in 2002 [31]. Since then, different variants of cache-side-channel attacks have been developed. These range from trace-based cache-side-channel attacks [1], to time-based cache-side-channel attacks [4], to access-based cache-side-channel attacks [30]. The latter category is most closely related to our work, because access-based cache-side-channel attacks have been used in combination with other micro-architectural features in transient execution attacks like [22].

Existing techniques for access-based cache-side-channel attacks include Prime+Probe [30], Evict+Time [30], Flush+Reload [45], and Flush+Flush [17]. Access-based attacks have been mounted, e.g., to recover cryptographic keys from OpenSSL AES [30], GnuPG RSA [45], and BouncyCastle AES [26].

6.2 Program Analysis for Cache Side Channels

Multiple approaches to both, qualitative and quantitative program analyses for the assessment of cache side channels exist.

Qualitative approaches include the tools CaSym [7], which targets cache-side channels in LLVM code, CacheS [41], which targets cache-side-channels at the binary level, and CacheD [42]. Quantitative approaches include the tool CacheAudit [16] and its successors [5, 15, 28, 29, 43], as well as CHALICE [12] and Abacus [2].

There are also complementary experiment-based approaches to detecting cache-side-channel leakage, including the tool DATA [44].

6.3 Transient Execution Vulnerabilities

The first transient-execution vulnerabilities were Meltdown [27] and Spectre [22], discovered in 2018. The former exploits cache side channels in combination with out-of-order execution. The latter exploits cache side channels in combination with speculative execution in an instruction pipeline. In general, transient-execution attacks exploit that a microarchitectural feature, e.g., a pipeline with branch prediction, causes the execution of an instruction that would usually not be executed and that this instruction leaves traces in the state of another microarchitectural feature, e.g., the cache. A recent survey [10] categorizes transient-execution attacks into Spectre-type attacks, Meltdown-type attacks and LVI-type attacks.

Spectre-type attacks include, e.g., Spectre v1 [22] (also called Spectre-PHT) and Spectre v2 [22] (also called Spectre-BTB), and NetSpectre [33] (a remote variant of Spectre v1). Spectre-type attacks target transient execution resulting from induced mispredictions. For instance, Spectre v1 and v2 mistrain the branch prediction for the outcome of direct branches and for the target of indirect branches, respectively, to trigger a transient execution.

Meltdown-type attacks exploit transient execution that happens out-of-order between a faulty behavior and the raising of the corresponding fault. The original Meltdown [27] exploits transient execution before a fault raised by a failed permission check. Foreshadow [8] extends the scope of Meltdown-based attacks across

the boundareis of SGX enclaves. Attacks like ZombieLoad [32] are based on reading old data that should not be available anymore.

Finally, LVI-type attacks are based on so-called load-value injection [37]. They inject attacker data into a transient execution between a faulty memory access and the raising of the fault.

6.4 Program Analysis for Transient Execution

Multiple execution models and program analyses have been developed in the context of transient-execution vulnerabilities already. However, the focus has been on qualitative analyses that aim at the detection of vulnerabilities, and on automated mitigations. We are not aware of any existing approach that supports the quantification of Spectre-v1-like vulnerabilities. In the following, we provide an overview of existing execution models and analyses.

An early execution model that takes into account speculative execution was developed by Disselkoen, Jagadeesan, Jeffrey and Riley [14]. They model the semantics of programs under speculation by partially ordered multisets that represent different executions. The execution model InSpectre [18] captures speculative execution at the level of micro-instructions.

The analysis tool Pitchfork [11] checks for violations of a speculative constant-time property. The property is formalized based on a semantics that takes into account a three-stage instruction pipeline with out-of-order execution. A recent extension of the Jasmin framework [3] supports the verification of a speculative memory-safety property and a speculative constant-time property for crypto implementations in the Jasmin language. The tool Blade [39] detects and mitigates violations of a speculative constant-time property in WebAssembly. To reduce the performance cost of the mitigation, the tool uses a new type of primitive that allows one to stop speculative execution at a more fine-grained level than fence instructions.

Spectector [19, 20] is an analysis tool that verifies a speculative non-interference property. The tool is based on symbolic execution and targets x64 assembly programs. The tool SpecSafe [6], which is also based on symbolic execution, checks programs in LLVM IR for violations of a speculation-aware non-interference property.

Other tools include oo7 [40], which automatically inserts fence instructions to mitigate potential leakage.

None of the above-mentioned analyses provides quantitative upper bounds on the leakage through side channels that exploit the combination of a cache and execution pipeline.

7 CONCLUSION

In this report, we have presented the first static program analysis for the reliable quantification of cache side channels in pipelined executions with branch prediction and out-of-order execution.

We developed a formal model that captures the execution of a simple assembly language on an architecture with a cache and pipeline. We focused on a single-level fully associative cache and a pipeline with static branch prediction. Our pipeline model features four stages (fetch, dispatch, execute and commit), alongside a model of the components used for out-of-order execution (reservation stations and reorder buffer).

Our program analysis is based on a combination of abstract interpretation and information theory. It can be used to obtain reliable upper bounds on the leakage to attackers that exploit the combination of the pipeline and cache using synchronous access-based cache-side-channel attacks.

Our evaluation across different variants of Spectre code snippets shows that our analysis, for the first time, allows to compare code with respect to the level of security against cache-side-channel attacks under speculative execution.

We hope that our quantitative analysis will pave the way to a more flexible treatment of Spectre-like vulnerabilities by allowing a more informed reasoning where to apply mitigations like fence-insertion with high performance cost.

ACKNOWLEDGMENTS

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - SFB 1119 - 236615297.

REFERENCES

[1] O. Aciiçmez and Ç. K. Koç. 2006. Trace-Driven Cache Attacks on AES (Short Paper). In *Proceedings of the 8th International Conference on Information and Communications Security (ICICS) (LNCS 4307)*. Springer, 112–121.

[2] Q. Bao, Z. Wang, X. Li, J. R. Larus, and D. Wu. 2021. Abacus: Precise Side-Channel Analysis. In *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 797–809.

[3] G. Barthe, S. Cauligi, B. Grégoire, A. Koutsos, K. Liao, T. Oliveira, S. Priya, T. Rezk, and P. Schwabe. 2021. High-Assurance Cryptography in the Spectre Era. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (S&P)*. IEEE, 1884–1901.

[4] D. J. Bernstein. 2005. *Cache-timing attacks on AES*. Technical Report. University of Illinois at Chicago.

[5] N. Bindel, J. Buchmann, J. Krämer, H. Mantel, J. Schickel, and A. Weber. 2017. Bounding the cache-side-channel leakage of lattice-based signature schemes using program semantics. In *Postproceedings of the 10th International Symposium on Foundations & Practice of Security (FPS) (LNCS 10723)*. Springer, 225–241.

[6] R. Brotzman, D. Zhang, M. T. Kandemir, and G. Tan. 2021. SpecSafe: Detecting Cache Side Channels in a Speculative World. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 129:1–129:29.

[7] R. L. Brotzman, S. L. Liu, D. Zhang, G. Tan, and M. T. Kandemir. 2018. CaSym: Cache Aware Symbolic Execution for Side Channel Detection and Mitigation. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 505–521.

[8] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-order Execution. In *Proceedings of the 27th USENIX Conference on Security Symposium (USENIX Security)*. USENIX Association, 991–1008.

[9] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom. 2019. Fallout: Leaking Data on Meltdown-resistant CPUs. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*. ACM, 769–784.

[10] C. Canella, K. N. Khasawneh, and D. Gruss. 2020. The Evolution of Transient-Execution Attacks. In *Proceedings of the 30th Great Lakes Symposium on VLSI (GLSVLSI)*. ACM, 163–168.

[11] S. Cauligi, C. Disselkoen, K. v. Gleissenthall, D. Tullsen, D. Stefan, T. Rezk, and G. Barthe. 2020. Constant-Time Foundations for the New Spectre Era. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 913–926.

[12] S. Chattopadhyay, M. Beck, A. Rezine, and A. Zeller. 2019. Quantifying the Information Leakage in Cache Attacks via Symbolic Execution. *ACM Transactions on Embedded Computing Systems* 18, 1 (2019), 1–27.

[13] P. Cousot and R. Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL)*. ACM, 238–252.

[14] C. Disselkoen, R. Jagadeesan, A. Jeffrey, and J. Riely. 2019. The Code that Never Ran: Modeling Attacks on Speculative Evaluation. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 1238–1255.

[15] G. Doychev and B. Köpf. 2017. Rigorous Analysis of Software Countermeasures against Cache Attacks. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementations (PLDI)*. ACM, 406–421.

[16] G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke. 2015. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. *ACM Transactions on Information*

and System Security (TISSEC) 18, 1 (2015), 4:1–4:32.

[17] D. Gruss, C. Maurice, K. Wagner, and S. Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA) (LNCS 9721)*. Springer, 279–299.

[18] R. Guanciale, M. Balliu, and M. Dam. 2020. InSpectre: Breaking and Fixing Microarchitectural Vulnerabilities by Formal Analysis. In *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS)*. ACM, 1853–1869.

[19] M. Guarnieri, B. Köpf, J.-F. Morales, J. Reineke, and A. Sanchez. 2020. Spectector: Principled Detection of Speculative Information Flows. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 1–19.

[20] M. Guarnieri, B. Köpf, J. Reineke, and P. Vila. 2021. Hardware-Software Contracts for Secure Speculation. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (S&P)*. IEEE, 1868–1883.

[21] P. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proceedings of the 16th Annual International Cryptology Conference (CRYPTO) (LNCS 1109)*. Springer, 104–113.

[22] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 1–19.

[23] P. Kocher, J. Jaffe, and B. Jun. 1999. Differential Power Analysis. In *Proceedings of the 19th Annual International Cryptology Conference (CRYPTO) (LNCS 1666)*. Springer, 388–397.

[24] B. Köpf, L. Mauborgne, and M. Ochoa. 2012. Automatic Quantification of Cache Side-Channels. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV) (LNCS 7358)*. Springer, 564–580.

[25] B. Köpf and G. Smith. 2010. Vulnerability Bounds and Leakage Resilience of Blinded Cryptography under Timing Attacks. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium (CSF)*. IEEE Computer Society, 44–56.

[26] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security)*. USENIX Association, 549–564.

[27] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*. USENIX Association, 973–990.

[28] H. Mantel, L. Scheidel, T. Schneider, A. Weber, C. Weinert, and T. Weißmantel. 2020. RiCaSi: Rigorous Cache Side Channel Mitigation via Selective Circuit Compilation. In *Proceedings of the 19th International Conference on Cryptology and Network Security (CANS) (LNCS 12579)*. Springer, 505–525.

[29] H. Mantel, A. Weber, and B. Köpf. 2017. A Systematic Study of Cache Side Channels across AES Implementations. In *Proceedings of the 9th International Symposium on Engineering Secure Software and Systems (ESSoS) (LNCS 10379)*. Springer, 213–230.

[30] D. A. Osvik, A. Shamir, and E. Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Proceedings of the 6th Cryptographer's Track at the RSA Conference (CT-RSA) (LNCS 3860)*. Springer, 1–20.

[31] D. Page. 2002. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. *IACR Cryptology ePrint Archive* 2002, 169 (2002), 1–23.

[32] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*. ACM, 753–768.

[33] M. Schwarz, M. Schwarzl, M. Lipp, and D. Gruss. 2019. NetSpectre: Read Arbitrary Memory over Network. In *Proceedings of the 24th European Symposium on Research in Computer Security (ESORICS) (LNCS 11735)*. Springer, 279–299.

[34] G. Smith. 2009. On the Foundations of Quantitative Information Flow. In *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures (FOSSACS) (LNCS 5504)*. Springer, 288–302.

[35] J. E. Smith. 1981. A Study of Branch Prediction Strategies. In *Proceedings of the 8th Annual Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, 135–148.

[36] R. M. Tomasulo. 1967. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development* 11, 1 (1967), 25–33.

[37] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lippi, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens. 2020. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 54–72.

[38] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. 2019. RIDL: Rogue In-flight Data Load. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 88–105.

[39] M. Vassena, C. Disselkoen, K. von Gleissenthall, S. Cauligi, R. G. Kıcı, R. Jhala, D. Tullsen, and D. Stefan. 2021. Automatically Eliminating Speculative Leaks from Cryptographic Code with Blade. In *Proceedings of the 48th ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM, 49:1–49:30.

[40] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury. 2019. oo7: Low-overhead Defense against Spectre attacks via Program Analysis. *IEEE Transactions on Software Engineering* (2019), To appear.

[41] S. Wang, Y. Bao, X. Liu, P. Wang, D. Zhang, and . Wu. 2019. Identifying Cache-Based Side Channels through Secret-Augmented Abstract Interpretation. In *USENIX Security*. USENIX Association, 657–674.

[42] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu. 2017. CacheD: Identifying Cache-Based Timing Channels in Production Software. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*. USENIX Association, 235–252.

[43] A. Weber, O. Nikiforov, A. Sauer, J. Schickel, G. Alber, H. Mantel, and T. Walther. 2021. Cache-Side-Channel Quantification and Mitigation for Quantum Cryptography. In *Proceedings of the 26th European Symposium on Research in Computer Security (ESORICS)*. Springer, 235–256.

[44] S. Weiser, A. Zankl, R. Spreitzer, K. Miller, S. Mangard, and G. Sigl. 2018. DATA – Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*. USENIX Association, 603–620.

[45] Y. Yarom and K. Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*. USENIX Association, 719–732.

A ANALYSIS DETAILS

A.1 Analysis Details for p-mask

First Clock Cycle.

$$S_1{}^{\#} = \texttt{a-upd}(Init^{\#}) = \left(\lambda(pr, pi^{\#}).\ \begin{cases} (rs_0{}^{\#}, m_0{}^{\#}, c_0{}^{\#}) & \text{if } pr = pr_m \wedge \\ & \quad pi^{\#} = pi_1{}^{\#} \\ ((\lambda r.\{\}), (\lambda d.\{\}), (\lambda d.\{\})) & \text{otherwise} \end{cases}\right),$$

where $pi_1{}^{\#} = \left(\left(\lambda s.\ \begin{cases} i2 & \text{if } s = fet \\ i1 & \text{if } s = dis \\ \top & \text{otherwise} \end{cases}\right), \langle\rangle, \langle\rangle\right).$

Second Clock Cycle.

$$S_2{}^{\#} = \texttt{a-upd}^2(Init^{\#}) = \left(\lambda(pr, pi^{\#}).\ \begin{cases} (rs_0{}^{\#}, m_0{}^{\#}, c_0{}^{\#}) & \text{if } pr = pr_m \wedge \\ & \quad pi^{\#} = pi_2{}^{\#} \\ ((\lambda r.\{\}), (\lambda d.\{\}), (\lambda d.\{\})) & \text{otherwise} \end{cases}\right),$$

where

$$pi_2{}^{\#} = \left(\left(\lambda s.\ \begin{cases} i3 & \text{if } s = fet \\ i2 & \text{if } s = dis \\ \top & \text{otherwise} \end{cases}\right), \langle\ (i2, \langle\rangle, \langle\rangle)\ \rangle, \langle\ (i1, \langle\rangle, \langle\rangle)\ \rangle, \langle\ (i1, ebx, \{0\})\ \rangle\right)$$

Third Clock Cycle.

$$S_3{}^{\#} = \texttt{a-upd}^3(Init^{\#}) = \left(\lambda(pr, pi^{\#}).\ \begin{cases} (rs_0{}^{\#}, m_0{}^{\#}, c_0{}^{\#}) & \text{if } pr = pr_m \wedge \\ & \quad pi^{\#} = pi_3{}^{\#} \\ ((\lambda r.\{\}), (\lambda d.\{\}), (\lambda d.\{\})) & \text{otherwise} \end{cases}\right),$$

where

$$pi_3{}^{\#} = \left(\left(\lambda s.\ \begin{cases} i4 & \text{if } s = fet \\ i3 & \text{if } s = dis \\ i1 & \text{if } s = exe \\ \top & \text{otherwise} \end{cases}\right), \langle\ (i2, \langle\rangle, \langle\rangle)\ \rangle, \langle\ (i1, ebx, \{0\}),\ (i2, ecx, \{15\})\ \rangle\right)$$

Fourth Clock Cycle.

$$S_4{}^{\#} = \texttt{a-upd}^4(Init^{\#}) = \left(\lambda(pr, pi^{\#}).\ \begin{cases} (rs_0{}^{\#}, m_0{}^{\#}, c_0{}^{\#}) & \text{if } pr = pr_m \wedge \\ & \quad pi^{\#} = pi_4{}^{\#} \\ ((\lambda r.\{\}), (\lambda d.\{\}), (\lambda d.\{\})) & \text{otherwise} \end{cases}\right),$$

where

$$pi_4{}^{\#} = \left(\left(\lambda s.\ \begin{cases} i5 & \text{if } s = fet \\ i4 & \text{if } s = dis \\ i2 & \text{if } s = exe \\ i1 & \text{if } s = com \end{cases}\right), \langle\ (i3, \langle(\{d0\}, 4)\rangle, \langle(ebx, i1)\rangle)\ \rangle, \langle\ (i1, ebx, \{0\}),\ (i2, ecx, \{15\}),\ (i3, ebx, \{3\})\ \rangle\right)$$

Fifth Clock Cycle.

$$S_5{}^{\#} = \texttt{a-upd}^5(Init^{\#}) = \left(\lambda(pr, pi^{\#}).\ \begin{cases} (rs_0{}^{\#}, m_0{}^{\#}, c_0{}^{\#}) & \text{if } pr = pr_m \wedge \\ & \quad pi^{\#} = pi_5{}^{\#} \\ ((\lambda r.\{\}), (\lambda d.\{\}), (\lambda d.\{\})) & \text{otherwise} \end{cases}\right),$$

where

$$pi_5{}^{\#} = \left(\left(\lambda s.\ \begin{cases} i6 & \text{if } s = fet \\ i5 & \text{if } s = dis \\ \top & \text{if } s = exe \\ i2 & \text{if } s = com \end{cases}\right), \langle\ (i3, \langle(\{d0\}, 3)\rangle, \langle(ebx, i1)\rangle),\ (i4, \langle\rangle, \langle(eax, i0), (ebx, i3)\rangle)\ \rangle, \langle\ (i2, ecx, \{15\}),\ (i3, ebx, \{3\})\ \rangle\right)$$

Sixth Clock Cycle.

$$S_6{}^{\#} = \texttt{a-upd}^6(Init^{\#}) = \left(\lambda(pr, pi^{\#}).\ \begin{cases} (rs_6{}^{\#}, m_0{}^{\#}, c_0{}^{\#}) & \text{if } pr = pr_m \wedge \\ & \quad pi^{\#} = pi_6{}^{\#} \\ ((\lambda r.\{\}), (\lambda d.\{\}), (\lambda d.\{\})) & \text{otherwise} \end{cases}\right),$$

where

$$pi_6{}^{\#} = \left(\left(\lambda s.\ \begin{cases} i7 & \text{if } s = fet \\ i6 & \text{if } s = dis \\ \top & \text{if } s = exe \\ \top & \text{if } s = com \end{cases}\right), \langle\ (i3, \langle(\{d0\}, 2)\rangle, \langle(ebx, i1)\rangle),\ (i4, \langle\rangle, \langle(eax, i0), (ebx, i3)\rangle),\ (i5, \langle(\{d4\}, 1)\rangle, \langle(eax, i0)\rangle)\ \rangle, \langle\ (i3, ebx, \{3\}),\ (i5, ecx, \{v \in Vals \mid 0 \le v \le 15\})\ \rangle\right)$$

and

$$rs_6{}^{\#} = \left(\lambda r.\ \begin{cases} \{3\} & \text{if } r = eax \\ \{15\} & \text{if } r = ecx \\ \{0\} & \text{otherwise} \end{cases}\right)$$

Seventh Clock Cycle.

$$S_7{}^{\#} = \texttt{a-upd}^7(Init^{\#}) = \left(\lambda(pr, pi^{\#}).\ \begin{cases} (rs_6{}^{\#}, m_0{}^{\#}, c_0{}^{\#}) & \text{if } pr = pr_m \wedge \\ & \quad pi^{\#} = pi_7{}^{\#} \\ ((\lambda r.\{\}), (\lambda d.\{\}), (\lambda d.\{\})) & \text{otherwise} \end{cases}\right),$$

where

$$pi_7{}^{\#} = \left(\left(\left(\lambda s.\ \begin{cases} \top & \text{if } s = fet \\ i7 & \text{if } s = dis \\ \top & \text{if } s = exe \\ \top & \text{if } s = com \end{cases}\right),\right.\right.$$
$$\langle\ (i3, \langle(\{d0\}, 1)\rangle, \langle(ebx, i1)\rangle),\ (i4, \langle\rangle, \langle(eax, i0), (ebx, i3)\rangle),$$
$$(i5, \langle(\{d4\}, 0)\rangle, \langle(eax, i0)\rangle),$$
$$(i6, \langle\{\mathsf{dadd}(d6, v) \mid 0 \le v \le 15\}, 4)\rangle, \langle(ecx, i5)\rangle)\ \rangle,$$
$$\langle\ (i3, ebx, \{3\}),$$
$$(i5, ecx, \{v \in Vals \mid 0 \le v \le 15\}),\ (i6, ecx, \{0\})\ \rangle)$$

Eighth Clock Cycle.

$$S_8{}^{\#} = \mathsf{a\text{-}upd}^8(Init^{\#}) =$$
$$\left(\lambda(pr, pi^{\#}).\ \begin{cases} (rs_6{}^{\#}, m_0{}^{\#}, c_8{}^{\#}) & \text{if } pr = pr_m \wedge \\ & \quad pi^{\#} = pi_8{}^{\#} \\ ((\lambda r.\{\}), (\lambda d.\{\}), (\lambda d.\{\})) & \text{otherwise} \end{cases}\right),$$

where

$$pi_8{}^{\#} = \left(\left(\left(\lambda s.\ \begin{cases} \top & \text{if } s = fet \\ \top & \text{if } s = dis \\ i5 & \text{if } s = exe \\ \top & \text{if } s = com \end{cases}\right),\right.\right.$$
$$\langle\ (i3, \langle(\{d0\}, 0)\rangle, \langle(ebx, i1)\rangle),\ (i4, \langle\rangle, \langle(eax, i0), (ebx, i3)\rangle),$$
$$(i6, \langle\{\mathsf{dadd}(d6, v) \mid 0 \le v \le 15\}, 3)\rangle, \langle(ecx, i5)\rangle)\ \rangle,$$
$$\langle\ (i3, ebx, \{3\}),$$
$$(i5, ecx, \{v \in Vals \mid 0 \le v \le 15\}),\ (i6, ecx, \{0\})\ \rangle)$$

and

$$c_8{}^{\#} = \left(\lambda d.\ \begin{cases} \{0\} & \text{if } d = d0 \\ \{1\} & \text{if } d = d4 \\ \{513\} & \text{otherwise} \end{cases}\right)$$

Ninth Clock Cycle.

$$S_9{}^{\#} = \mathsf{a\text{-}upd}^9(Init^{\#}) =$$
$$\left(\lambda(pr, pi^{\#}).\ \begin{cases} (rs_6{}^{\#}, m_0{}^{\#}, c_8{}^{\#}) & \text{if } pr = pr_m \wedge \\ & \quad pi^{\#} = pi_9{}^{\#} \\ ((\lambda r.\{\}), (\lambda d.\{\}), (\lambda d.\{\})) & \text{otherwise} \end{cases}\right),$$

where

$$pi_9{}^{\#} = \left(\left(\left(\lambda s.\ \begin{cases} \top & \text{if } s = fet \\ \top & \text{if } s = dis \\ i3 & \text{if } s = exe \\ i5 & \text{if } s = com \end{cases}\right),\right.\right.$$
$$\langle\ (i4, \langle\rangle, \langle(eax, i0), (ebx, i3)\rangle),$$
$$(i6, \langle\{\mathsf{dadd}(d6, v) \mid 0 \le v \le 15\}, 2)\rangle, \langle(ecx, i5)\rangle)\ \rangle,$$
$$\langle\ (i3, ebx, \{3\}),$$
$$(i5, ecx, \{v \in Vals \mid 0 \le v \le 15\}),\ (i6, ecx, \{0\})\ \rangle)$$

Tenth Clock Cycle.

$$S_{10}{}^{\#} = \mathsf{a\text{-}upd}^{10}(Init^{\#}) =$$
$$\left(\lambda(pr, pi^{\#}).\ \begin{cases} (rs_{10}{}^{\#}, m_0{}^{\#}, c_8{}^{\#}) & \text{if } pr = pr_m \wedge \\ & \quad pi^{\#} = pi_{10}{}^{\#} \\ ((\lambda r.\{\}), (\lambda d.\{\}), (\lambda d.\{\})) & \text{otherwise} \end{cases}\right),$$

where

$$pi_{10}{}^{\#} = \left(\left(\left(\lambda s.\ \begin{cases} \top & \text{if } s = fet \\ \top & \text{if } s = dis \\ i4 & \text{if } s = exe \\ i3 & \text{if } s = com \end{cases}\right),\right.\right.$$
$$\langle\ (i4, \langle\rangle, \langle(eax, i0), (ebx, i3)\rangle),$$
$$(i6, \langle\{\mathsf{dadd}(d6, v) \mid 0 \le v \le 15\}, 1)\rangle, \langle(ecx, i5)\rangle)\ \rangle,$$
$$\langle\ (i3, ebx, \{3\}),\ (i6, ecx, \{0\})\ \rangle)$$

and

$$rs_{10}{}^{\#} = \left(\lambda r.\ \begin{cases} \{3\} & \text{if } r = eax \\ \{v \in Vals \mid 0 \le v \le 15\} & \text{if } r = ecx \\ \{0\} & \text{otherwise} \end{cases}\right)$$

Eleventh Clock Cycle.

$$S_{11}{}^{\#} = \mathsf{a\text{-}upd}^{11}(Init^{\#}) =$$
$$\left(\lambda(pr, pi^{\#}).\ \begin{cases} (rs_{11}{}^{\#}, m_0{}^{\#}, c_{11}{}^{\#}) & \text{if } pr = pr_m \wedge \\ & \quad pi^{\#} = pi_{11}{}^{\#} \\ ((\lambda r.\{\}), (\lambda d.\{\}), (\lambda d.\{\})) & \text{otherwise} \end{cases}\right),$$

where

$$pi_{11}{}^{\#} = \left(\left(\left(\lambda s.\ \begin{cases} i7 & \text{if } s = fet \\ \top & \text{if } s = dis \\ \top & \text{if } s = exe \\ \top & \text{if } s = com \end{cases}\right), \langle\rangle, \langle\rangle\right)\right.$$

and

$$rs_{11}{}^{\#} = \left(\lambda r.\ \begin{cases} \{3\} & \text{if } r \in \{eax, ebx\} \\ \{v \in Vals \mid 0 \le v \le 15\} & \text{if } r = ecx \\ \{0\} & \text{otherwise} \end{cases}\right)$$

and

$$c_{11}{}^{\#} = \left(\lambda d.\ \begin{cases} \{1\} & \text{if } d = d0 \\ \{2\} & \text{if } d = d4 \\ \{0, 514\} & \text{if } d \in \{\mathsf{dadd}(d6, v) \mid 0 \le v \le 15\} \\ \{514\} & \text{otherwise} \end{cases} \right)$$

Twelfth Clock Cycle.

$$S_{12}{}^{\#} = \mathsf{a\text{-}upd}^{12}(Init^{\#}) = \left(\lambda(pr, pi^{\#}).\ \begin{cases} (rs_{11}{}^{\#}, m_0{}^{\#}, c_{11}{}^{\#}) & \text{if } pr = pr_m \wedge \\ & \quad pi^{\#} = pi_{12}{}^{\#} \\ ((\lambda r.\{\}), (\lambda d.\{\}), (\lambda d.\{\})) & \text{otherwise} \end{cases} \right),$$

where

$$pi_{12}{}^{\#} = \left(\left(\lambda s.\ \begin{cases} \top & \text{if } s = fet \\ i7 & \text{if } s = dis \\ \top & \text{if } s = exe \\ \top & \text{if } s = com \end{cases} \right), \langle\rangle, \langle\rangle \right)$$

Thirteenth Clock Cycle.

$$S_{13}{}^{\#} = \mathsf{a\text{-}upd}^{13}(Init^{\#}) = \left(\lambda(pr, pi^{\#}).\ \begin{cases} (rs_{11}{}^{\#}, m_0{}^{\#}, c_{11}{}^{\#}) & \text{if } pr = pr_m \wedge \\ & \quad pi^{\#} = pi_{13}{}^{\#} \\ ((\lambda r.\{\}), (\lambda d.\{\}), (\lambda d.\{\})) & \text{otherwise} \end{cases} \right),$$

where

$$pi_{13}{}^{\#} = \left(\left(\lambda s.\ \begin{cases} \top & \text{if } s = fet \\ \top & \text{if } s = dis \\ \top & \text{if } s = exe \\ \top & \text{if } s = com \end{cases} \right), \langle (i7, \langle\rangle, \langle\rangle) \rangle, \langle\rangle \right)$$

Fourteenth Clock Cycle.

$$S_{14}{}^{\#} = \mathsf{a\text{-}upd}^{14}(Init^{\#}) = \left(\lambda(pr, pi^{\#}).\ \begin{cases} (rs_{11}{}^{\#}, m_0{}^{\#}, c_{11}{}^{\#}) & \text{if } pr = pr_m \wedge \\ & \quad pi^{\#} = pi_{14}{}^{\#} \\ ((\lambda r.\{\}), (\lambda d.\{\}), (\lambda d.\{\})) & \text{otherwise} \end{cases} \right),$$

where

$$pi_{14}{}^{\#} = \left(\left(\lambda s.\ \begin{cases} \top & \text{if } s = fet \\ \top & \text{if } s = dis \\ i7 & \text{if } s = exe \\ \top & \text{if } s = com \end{cases} \right), \langle\rangle, \langle\rangle \right)$$

Fifteenth Clock Cycle.

$$S_{15}{}^{\#} = \mathsf{a\text{-}upd}^{15}(Init^{\#}) = \left(\lambda(pr, pi^{\#}).\ \begin{cases} (rs_{11}{}^{\#}, m_0{}^{\#}, c_{11}{}^{\#}) & \text{if } pr = pr_m \wedge \\ & \quad pi^{\#} = pi_{15}{}^{\#} \\ ((\lambda r.\{\}), (\lambda d.\{\}), (\lambda d.\{\})) & \text{otherwise} \end{cases} \right),$$

where

$$pi_{15}{}^{\#} = \left(\left(\lambda s.\ \begin{cases} \top & \text{if } s = fet \\ \top & \text{if } s = dis \\ \top & \text{if } s = exe \\ i7 & \text{if } s = com \end{cases} \right), \langle\rangle, \langle\rangle \right)$$

Sixteenth Clock Cycle.

$$S_{16}{}^{\#} = \mathsf{a\text{-}upd}^{16}(Init^{\#}) = \left(\lambda(pr, pi^{\#}).\ \begin{cases} (rs_{11}{}^{\#}, m_0{}^{\#}, c_{11}{}^{\#}) & \text{if } pr = pr_m \wedge \\ & \quad pi^{\#} = pi_{16}{}^{\#} \\ ((\lambda r.\{\}), (\lambda d.\{\}), (\lambda d.\{\})) & \text{otherwise} \end{cases} \right),$$

where

$$pi_{16}{}^{\#} = \left(\left(\lambda s.\ \begin{cases} \top & \text{if } s = fet \\ \top & \text{if } s = dis \\ \top & \text{if } s = exe \\ \top & \text{if } s = com \end{cases} \right), \langle\rangle, \langle\rangle \right)$$

Seventeenth Clock Cycle.

$$S_{17}{}^{\#} = S_{16}{}^{\#}$$

A.2 Analysis Details for `p-shift`

The programs pr_m and pr_s differ only in instruction i5. The analysis proceeds analogously to the analysis of `p-mask`, where pr_m is replaced by pr_s. The only differences are:

$$rs_{10}{}^{\#\prime} = \left(\lambda r.\ \begin{cases} \{3\} & \text{if } r = eax \\ \{v \in Vals \mid & \text{if } r = ecx \\ \quad 0 \le v \le 131071\} & \\ \{0\} & \text{otherwise} \end{cases} \right)$$

$$rs_{11}{}^{\#\prime} = \left(\lambda r.\ \begin{cases} \{3\} & \text{if } r \in \{eax, ebx\} \\ \{v \in Vals \mid & \text{if } r = ecx \\ \quad 0 \le v \le 131071\} & \\ \{0\} & \text{otherwise} \end{cases} \right)$$

$$c_{11}{}^{\#\prime} = \left(\lambda d.\ \begin{cases} \{1\} & \text{if } d = d0 \\ \{2\} & \text{if } d = d4 \\ \{0, 514\} & \text{if } d \in \{\mathsf{dadd}(d6, v) \mid \\ & \qquad 0 \le v \le 131071\} \\ \{514\} & \text{otherwise} \end{cases} \right)$$

$$pi_6{}^{\#\prime} = \left(\left(\lambda s.\ \begin{cases} i7 & \text{if } s = fet \\ i6 & \text{if } s = dis \\ \top & \text{if } s = exe \\ \top & \text{if } s = com \end{cases} \right), \right.$$
$$\langle\ (i3, \langle(\{d0\}, 2)\rangle, \langle(ebx, i1)\rangle),\ (i4, \langle\rangle, \langle(eax, i0), (ebx, i3)\rangle),$$
$$\left. (i5, \langle(\{d4\}, 1)\rangle, \langle(eax, i0)\rangle)\ \rangle, \right.$$
$$\langle\ (i3, ebx, \{3\}),\ (i5, ecx, \{v \in Vals \mid 0 \le v \le 131071\})\ \rangle)$$

$$pi_7{}^{\#\prime} = \left\langle\left\langle \lambda s.\ \begin{cases} \top & \text{if } s = fet \\ i7 & \text{if } s = dis \\ \top & \text{if } s = exe \\ \top & \text{if } s = com \end{cases}\right\rangle,$$
$$\langle\ (i3, \langle(\{d0\}, 1)\rangle, \langle(ebx, i1)\rangle),\ (i4, \langle\rangle, \langle(eax, i0), (ebx, i3)\rangle),$$
$$(i5, \langle(\{d4\}, 0)\rangle, \langle(eax, i0)\rangle),$$
$$(i6, \langle\{\mathsf{dadd}(d6, v) \mid 0 \le v \le 131071\}, 4)\rangle, \langle(ecx, i5)\rangle)\ \rangle,$$
$$\langle\ (i3, ebx, \{3\}), (i5, ecx, \{v \in Vals \mid 0 \le v \le 131071\}),$$
$$(i6, ecx, \{0\})\ \rangle)$$

$$pi_8{}^{\#\prime} = \left\langle\left\langle \lambda s.\ \begin{cases} \top & \text{if } s = fet \\ \top & \text{if } s = dis \\ i5 & \text{if } s = exe \\ \top & \text{if } s = com \end{cases}\right\rangle,$$
$$\langle\ (i3, \langle(\{d0\}, 0)\rangle, \langle(ebx, i1)\rangle),\ (i4, \langle\rangle, \langle(eax, i0), (ebx, i3)\rangle),$$
$$(i6, \langle\{\mathsf{dadd}(d6, v) \mid 0 \le v \le 131071\}, 3)\rangle, \langle(ecx, i5)\rangle)\ \rangle,$$
$$\langle\ (i3, ebx, \{3\}), (i5, ecx, \{v \in Vals \mid 0 \le v \le 131071\}),$$
$$(i6, ecx, \{0\})\ \rangle)$$

$$pi_9{}^{\#\prime} = \left\langle\left\langle \lambda s.\ \begin{cases} \top & \text{if } s = fet \\ \top & \text{if } s = dis \\ i3 & \text{if } s = exe \\ i5 & \text{if } s = com \end{cases}\right\rangle,$$
$$\langle\ (i4, \langle\rangle, \langle(eax, i0), (ebx, i3)\rangle),$$
$$(i6, \langle\{\mathsf{dadd}(d6, v) \mid 0 \le v \le 131071\}, 2)\rangle, \langle(ecx, i5)\rangle)\ \rangle,$$
$$\langle\ (i3, ebx, \{3\}), (i5, ecx, \{v \in Vals \mid 0 \le v \le 131071\}),$$
$$(i6, ecx, \{0\})\ \rangle)$$

$$pi_{10}{}^{\#\prime} = \left\langle\left\langle \lambda s.\ \begin{cases} \top & \text{if } s = fet \\ \top & \text{if } s = dis \\ i4 & \text{if } s = exe \\ i3 & \text{if } s = com \end{cases}\right\rangle,$$
$$\langle\ (i4, \langle\rangle, \langle(eax, i0), (ebx, i3)\rangle),$$
$$(i6, \langle\{\mathsf{dadd}(d6, v) \mid 0 \le v \le 131071\}, 1)\rangle, \langle(ecx, i5)\rangle)\ \rangle,$$
$$\langle\ (i3, ebx, \{3\}),\ \ (i6, ecx, \{0\})\ \rangle)$$