Typing High-Speed Cryptography against Spectre v1

Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira
Swarn Priya, Peter Schwabe, Lucas Tabary-Maujean

MPI-SP, Bochum, Germany
IMDEA Software Institute, Madrid, Spain
Inria, Sophia Antipolis, France
Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France
ENS Paris-Saclay, Gif-sur-Yvette, France
Radboud University, Nijmegen, The Netherlands

***Abstract*—The current gold standard of cryptographic software is to write efficient libraries with systematic protections against timing attacks. In order to meet this goal, cryptographic engineers increasingly use high-assurance cryptography tools. These tools guide programmers and provide rigorous guarantees that can be verified independently by library users. However, high-assurance tools reason about overly simple execution models that elide transient execution leakage. Thus, implementations validated by high-assurance cryptography tools remain potentially vulnerable to transient execution attacks such as Spectre or Meltdown. Moreover, proposed countermeasures are not used in practice due to performance overhead.**

We propose, analyze, implement and evaluate an approach for writing efficient cryptographic implementations that are protected against Spectre v1 attacks. Our approach ensures speculative constant-time, an information flow property which guarantees that programs are protected against Spectre v1. Speculative constant-time is enforced by means of a (value-dependent) information flow type system. The type system tracks security levels depending on whether execution is misspeculating. We implement our approach in the Jasmin framework for high-assurance cryptography, and use it for protecting all implementations of an experimental cryptographic library that includes highly optimized implementations of symmetric primitives, of elliptic-curve cryptography, and of Kyber, a lattice-based KEM recently selected by NIST for standardization. The performance impact of our protections is very low; for example, less than 1% for Kyber and essentially zero for X25519.

1. Introduction

Modern applications use cryptographic libraries extensively, generally under the expectation that these libraries incur minimal performance overhead and provide strong correctness and security guarantees. While these expectations are fully reasonable, cryptographic engineering is intrinsically hard, and principled approaches are required to build better libraries. High-assurance cryptography [1] is an emerging approach that emphasizes foundational approaches to cryptographic engineering. To date, high-assurance cryptography has successfully delivered methods and tools for building efficient implementations with strong guarantees of functional correctness, provable security, and protection against cache-based timing attacks. In turn, these methods and tools have been used to develop high-assurance cryptographic software which has been deployed widely in mainstream software [2], [3].

These successes show the benefits of high-assurance cryptography at this point of time. However, new standards emerge, and attacks only get better, which requires to broaden the scope and the standards of high-assurance cryptography. In this paper, we consider two major developments:

- transient execution attacks: Spectre [4] and Meltdown [5] demonstrated that micro-architectural side-channels can have devastating effects;
- post-quantum cryptography: the NIST Post-Quantum Competition [6] recently selected four cryptographic algorithms to protect against quantum computers.

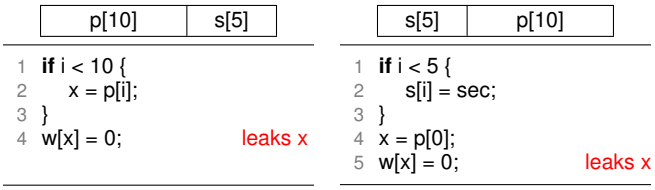
Motivated by these developments, we consider the problem of helping cryptographic engineers write efficient, Spectre-protected, implementations of post-quantum algorithms selected by NIST, so that migration to these new primitives also comes with a higher level of protection against implementation attacks. In this work, we focus specifically on protecting implementations against Spectre v1 attacks. These attacks are an instance of Spectre attacks that exploit branch prediction. In contrast to other forms of Spectre attacks, it is commonly agreed that critical software must deploy countermeasures to mitigate against Spectre v1. However, writing cryptographic implementations that are protected against Spectre v1 *and* achieve similar performance is far from immediate. On the one hand, protections based on fences incur a significant performance overhead. On the other hand, more efficient protections based on (value or

address) hardening, including all forms of Speculative Load Hardening [7], are difficult to apply. The main reason for this is that high-speed implementations typically use the available resources as much as possible. This means that register usage (and registers are available on a limited supply) is maximized as much as possible to avoid as many loads and stores from memory as possible. Using as many registers as possible directly correlates with speed, and it can also be related to reducing the code size under certain circumstances, considering that fewer instructions are needed. However, in order to be compatible with protections based on hardening, the implementations might need to be rewritten to free one register for the misspeculation flag, which is used to ensure that masking is only applied to speculated values.

Problem statement and contributions. The main high-level contribution of this paper is an approach that provably protects cryptographic implementations against Spectre v1 attacks with minimal performance overhead. For example, the cost of protecting a highly-optimized AVX2 implementation of Kyber768 against Spectre v1 is less than 1%.

To achieve this goal, we make the following technical contributions:

- We define core primitives that support fine-grained protection against Spectre v1 attacks. Our primitives provide an interface to update and use misspeculation flags. These flags can be used by programmers to track whether program execution is normal, or misspeculating. We demonstrate how our primitives can encode both compiler-level mitigations, including Speculative Load Hardening [7] and Selective Speculative Load Hardening [8], and algorithm-specific mitigations.
- We define a type system to track that primitives are correctly used. Technically, our type system uses a value-dependent information flow analysis, where security levels depend on the misspeculation flag. In order to enforce this dependency, our type system additionally maintains a misspeculation typestate, which tracks how the misspeculation flag is stored in registers. We show that our type system is *sound*: typable programs are speculative constant-time, which generalizes constant-time to the speculative setting [9]. In addition, we also consider a more precise type system which uses an external checker to distinguish between safe and unsafe memory accesses. We consider out-of-bound access as unsafe access.
- We use our approach to protect with minimal performance overhead reference and hand-optimized vectorized implementations of several cryptographic algorithms, including symmetric primitives like ChaCha20 [10] and Poly1305 [11]; the X25519 elliptic-curve key exchange [12]; and Kyber [13]. Concretely, we implement our approach in the Jasmin framework for high-assurance cryptography [14], [15], and modify existing implementations so that they can be type-checked by our system. As previously explained, this modification goes beyond inserting protection primitives, and is achieved through a combination of code refactoring, primarily instruction scheduling and spilling, and *ad hoc* techniques, notably the use of mmx registers to keep public values out of the stack.



(a) v1-read (where i = 10) (b) v1-write (where i = 5)

Figure 1: Spectre v1 gadgets. Array p and register x are public and array s and value sec are secret.

Supplementary material. Our implementation of the type system and modified libraries are available at https://artifacts.formosa-crypto.org/data/selslh.tar.bz2.

2. Preliminaries

Constant-time programming. The *constant-time (CT) policy* is an information-flow policy to protect against cache-based timing attacks. The policy mandates that control-flow and memory accesses be independent of secret data, and can be enforced by means of an information flow type system. However, the CT policy only provides guarantees if programs are executed in simple microprocessors *without* speculation.

Spectre v1 attacks. Modern microprocessors aggressively use branch, address and value predictions to proceed with execution without awaiting for the result of prior computations. After some time, the real value must have been computed and must be checked against the guessed value. In case of an incorrect guess, the results of erroneous computations are simply discarded and execution is rolled back to continue with the correct value. However, the effects of the speculative execution on the micro-architectural state remain after rollback.

Spectre [4] is a class of cache-based timing attacks which exploit this persistent effect to retrieve secret data from program execution. An essential aspect of a practical Spectre attack is to train the predictor so that it achieves a desired behavior. However, for the purpose of this work, it suffices to consider a conservative threat model where attackers have control over the predictor.

In this paper, we focus on Spectre v1 attacks, which are based on branch prediction. Figure 1a presents a Spectre gadget that leaks secret data speculatively via branch misprediction. Suppose that the initial value of i is 10. Under standard execution, the branch will not be taken, and therefore the program does not leak. However, the branch may be taken under speculative execution, if the branch predictor incorrectly guesses that i < 10 evaluates to true. In this case, execution will read past the end of the array p: the secret value contained at position s[0] is loaded into variable x at line 2; in other words, x becomes *transient*. In this paper, we call *transient* a security type where a public variable may

speculatively depend on a secret. The instruction at line 4 accesses array w at address x and leaks s[0].

Figure 1b presents another Spectre gadget that leaks secret data speculatively via branch misprediction. Suppose that the initial value of i is 5. In this case, branch misprediction will cause execution to write past the end of array p: sec will be speculatively stored at position p[0] (line 1). Then the load at line 4 will load sec into x, and the store instruction at line 5 will leak sec.

Note that both programs are constant-time and that in both cases, the processor will eventually detect that the wrong branch was taken and will backtrack. Still, it is too late: the leak has occurred, concretely, the state of the cache depends on secret data.

Countermeasures. There are many ways to protect CT programs against Spectre v1 attacks. One simple countermeasure is to insert a fence instruction immediately after each branching instruction. However, this countermeasure incurs a significant overhead. A more efficient alternative is to insert enough fence instructions so that there is always a fence between the moment when a register becomes transient and the moment when it leaks via control-flow or memory accesses [16]. This still ensures that programs are speculative constant-time, but with a lower performance overhead. However, it is generally better to minimize the use of fences. The LLVM compiler implements one such alternative, called Speculative Load Hardening (SLH) [7]. The key idea of SLH is to mask speculatively loaded values with a constant. This masking is conditioned on a *misspeculation flag* which indicates whether or not the program's execution is being misspeculated. The flag is initialized at the beginning of the program's execution alongside a lfence instruction to ensure that the program is initially executed non-speculatively. In addition, the flag is updated on each branch's entry and exit points (for instance, while loops or if statements). Using the misspeculation flag, SLH masks speculatively loaded values by a default value. SLH is often (but not always) more efficient than inserting fences. However, one can further minimize overhead by only masking transient values that are assigned to public registers. To our best knowledge, this optimization, called Selective Speculative Load Hardening [8], has not been implemented before.

3. Programming hardened implementations

3.1. Threat model and security notion

Our threat model assumes that the attacker can observe branching decisions and addresses (not values) of memory accesses and fully controls the branch predictor, the targets of all unsafe reads and the values of all unsafe writes. Our threat model errs on the conservative side; as a result, it may not always be obvious to mount practical attacks against programs that are insecure in our model. However, we are concerned with protecting programs, therefore strong threat models are to be preferred, as long as they do not overly affect performance of protected programs.

$$
\begin{array}{rcl}
e & := & x \mid o(\vec{x}) \\
i & := & x = e \mid x = a[e] \mid a[e] = x \\
 & \mid & \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid \mathsf{while}\ e\ \mathsf{do}\ c \\
 & \mid & \mathsf{ms} = \mathsf{init_msf}() \mid \mathsf{ms} = \mathsf{set_msf}(e, \mathsf{ms}) \\
 & \mid & x_1 = \mathsf{protect}(x_2, \mathsf{ms}) \\
c & := & \epsilon \mid i; c
\end{array}
$$

Figure 2: Core language with protection primitives. x ranges over registers and a ranges over arrays.

The security notion attached to our threat model is *speculative constant-time*. Informally, a program is speculative constant time if its leakage does not depend on secrets, for every attacker's choice of branch decisions and unsafe memory accesses.

3.2. Language

Figure 2 presents a core language with instructions for assignments, loads, stores, conditionals, and loops. The language additionally features three primitives:

- ms = init_msf () sets the misspeculation flag ms to 0. This primitive is typically used at the beginning of a program and it also acts as a serializing instruction;
- ms = set_msf (e, ms) updates the misspeculation flag. This primitive may be used immediately after a branching instruction conditioned on e. However, it is sometimes possible to postpone updating the flag, or even not to update it, without compromising security;
- x = protect (x, ms) conditionally masks the register x depending on the value of ms. Specifically, the value of x remains unchanged in a correct execution and is set to -1 in a misspeculated execution.

Semantics. The behavior of programs is traditionally modeled using a labeled transition relation of the form $c, s \to c', s'$, where c and c' are programs, s and s' are states. This relation states that one step of execution of the program c starting from state s leads to state s', with continuation c'. This relation is adequate to reason about functional correctness of programs under a sequential semantics, but is not appropriate to reason about leakage of programs under a speculative semantics. For this purpose, we enrich the classic transition relation and consider judgments of the form $c, \langle s, b\rangle \xrightarrow[d]{o} c', \langle s', b'\rangle$ where o is an observation, d is a directive, and b and b' represent the value of the misspeculation flag.

The misspeculation flag tracks whether execution is misspeculating. By convention, the flag is set to true when execution is misspeculating, and false otherwise.

The directive d represents the attacker's decision before the instruction is executed, whereas the observation o represents the knowledge gained by the attacker after the instruction is executed. Formally, the sets of directives and observations are defined by the grammars:

$$
\begin{array}{rcl}
d \in \mathsf{Dir} & ::= & \mathsf{step} \mid \mathsf{force} \mid \mathsf{load}\ a, i \mid \mathsf{store}\ a, i \\
o \in \mathsf{Obs} & ::= & \bullet \mid \mathsf{read}\ a, v \mid \mathsf{write}\ a, v \mid \mathsf{branch}\ b
\end{array}
$$

The directive step is used by the attacker to let execution proceed normally. When such a directive is issued on a branching instruction, the guard is evaluated w.r.t. the current state, and the program enters in the correct branch. The directive force is issued by the attacker to force execution to enter into a misspeculated branch. The force directive is the only directive that modifies the misspeculation flag. Finally, the directives load a, i and store a, i allow the attacker to read from and write to addresses of its choice in case of a (speculatively) unsafe memory read or write.

The observation $\bullet$ leaks no information. The observation read a, v and write a, v leak the address of memory accesses. Finally, the observation branch b leaks the value of the guard when executing a branching statement.

The formal definition of states and execution coincides with [8] and is given in Appendix B. For the purpose of this section, it suffices to state the formal definition of speculative constant-time in our setting. We write $c, \langle s, b\rangle \xrightarrow[D]{O}\!\!\!\!\rightarrow c', \langle s', b'\rangle$ for the reflexive-transitive closure of one-step execution.

Definition 1 (Speculative constant-time)**.** *Let ϕ be an equivalence relation on states. A program c is speculative constant-time with respect to ϕ (or ϕ-SCT), iff for every set of directives D, every pair of initial states s_1 and s_2 such that $s_1 \ \phi \ s_2$, and every pair of executions $c, \langle s_1, \bot\rangle \xrightarrow[D]{O_1}\!\!\!\!\rightarrow c_1', \langle s_1', b_1'\rangle$ and $c, \langle s_2, \bot\rangle \xrightarrow[D]{O_2}\!\!\!\!\rightarrow c_2', \langle s_2', b_2'\rangle$, we have $O_1 = O_2$.*

Our definition only requires equality of observations—which in itself suffices to ensure absence of leakage. However, our proofs use a stronger notion that additionally enforce that output configurations are related by a suitable relation.

Speculative constant-time implies constant-time. The latter corresponds to the case where D contains no force directive.

3.3. Examples

We now illustrate how our primitives can be used to implement countermeasures and algorithm-specific protections which guarantee speculative constant-time.

Speculative Load Hardening. Figure 3a illustrates how to use the aforementioned primitives to implement Speculative Load Hardening of the program of Figure 1a. The initialization instruction at line 1 ensures that the code is entered in normal execution mode, and with the misspeculation flag ms set to 0. The flag-updating set_msf instructions at lines 4 and 8 ensure that the ms register is maintained up-to-date after entering the branches. Finally the protect instruction at line 6 ensures that the value of x is speculatively masked. Together the inserted instructions ensure that the program produces the expected result under normal execution and do not leak secret values under speculative execution, since x is set to -1 if execution is misspeculated. Similarly, Figure 3b protects Figure 1b.

Selective Speculative Load Hardening. Consider the one-time-pad program Figure 4a, which performs secret loads from

```
1  ms = init_msf();
2  b = i < 10;
3  if b {
4      ms = set_msf(b, ms);
5      x = p[i];
6      x = protect(x, ms);
7  } else {
8      ms = set_msf(! b, ms);
9  }
10 w[x] = 0;
11
```

(a) Protected v1-read

```
1  ms = init_msf();
2  b = i < 5;
3  if b {
4      ms = set_msf(b, ms);
5      s[i] = secret;
6  } else {
7      ms = set_msf(! b, ms);
8  }
9  x = p[0];
10 x = protect(x, ms);
11 w[x] = 0;
```

(b) Protected v1-write

Figure 3: Protected v1 Spectre gadgets

msg and key. Figure 4b and Figure 4c show implementations protected by SLH and selSLH respectively. SLH protects all the loads and selSLH does not protect as they are not public loads. However, both maintain the misspeculation flag by updating ms for future use.

Algorithm-specific protections. Figure 5a accumulates the ten first elements of a public array p and stores the result in a register s, using a local variable t. Note that speculative execution may cause the load instruction in line 3 to write secret values into t, and hence the result s is transient, i.e., it may speculatively depend on secrets. The remaining programs in Figure 5 are different protected versions which allow to make s public at the end of the program.

Figure 5b is the protected version obtained by applying selSLH, i.e., systematically updating the misspeculation flag and masking public loaded values. Therefore the register ms is updated according to the control flow (lines 4 and 10) and loaded values are protected (line 6) and only public values flow into s.

Figure 5c is a protected version that uses a single protect. In this case, s remains transient during execution of the loop. However, s is never used, so it suffices to protect it at the end of the function. Note that the ms variable is still updated at each iteration.

Figure 5d is a protected version that updates the misspeculation flag only once at line 8 with the condition i == 10. This tests suffices to detect misspeculation: since the loop body does not have control-flow, the only form of misspeculation is to iterate the while loop for i times, with $i \neq 10$.

No protection. Figure 6 shows two programs that do not need protection. For both programs, we assume that array p and value pub_v are public, while array s and value sec_v are secret, and we require that x be public at the end of the execution. The programs are memory-safe in normal execution.

Consider the program from Figure 6a. Although speculative execution of the branch at line 4 may result in storing the value pub_v out-of-bounds, we still know that the value x at line 8 will be public (it is either the normal value p[5] or the speculative value pub_v). Moreover the value stored does not depend on any secret. Hence, x does not need to be protected.

```
1  i = 0;
2  while (i < 8) {
3      t1 = msg[i];
4      t2 = key[i];
5      t1 ^= t2;
6      msg[i] = t1;
7      i += 1;
8  }
```

(a) unprotected

```
1  ms = init_ms();
2  i = 0;
3  while (i < 8) {
4      ms = set_msf(i < 8, ms);
5      t1 = msg[i];
6      t1 = protect(t1, ms);
7      t2 = key[i];
8      t2 = protect(t2, ms);
9      t1 ^= t2;
10     msg[i] = t1;
11     i += 1;
12 }
13 ms = set_msf(i >= 8, ms);
```

(b) SLH

```
1  ms = init_ms();
2  i = 0;
3  while (i < 8) {
4      ms = set_msf(i < 8, ms);
5      t1 = msg[i];
6      t2 = key[i];
7      t1 ^= t2;
8      msg[i] = t1;
9      i += 1;
10 }
11 ms = set_msf(i >= 8, ms);
```

(c) selSLH

Figure 4: Protected one-time pad

```
1  s = 0; i = 0;
2  while (i < 10) {
3      t = p[i];
4      s += t;
5      i += 1;
6  }
```

(a) Unprotected

```
1  ms = init_ms();
2  s = 0; i = 0;
3  while (i < 10) {
4      ms = set_msf(i < 10, ms);
5      t = p[i];
6      t = protect(t, ms);
7      s += t;
8      i += 1;
9  }
10 ms = set_msf(i >= 10, ms);
```

(b) Protect each iteration

```
1  ms = init_ms();
2  s = 0; i = 0;
3  while (i < 10) {
4      ms = set_msf(i < 10, ms);
5      t = p[i];
6      s += t;
7      i += 1;
8  }
9  ms = set_msf(i >= 10, ms);
10 s = protect(s, ms);
```

(c) Protect final value

```
1  ms = init_ms();
2  s = 0; i = 0;
3  while (i < 10) {
4      t = p[i];
5      s += t;
6      i += 1;
7  }
8  ms = set_msf(i == 10, ms);
9  s = protect(s, ms);
```

(d) Single update of msf

Figure 5: Protected Sum

```
1  ms = init_ms();
2  if b {
3      ms = set_msf(b, ms);
4      s[i] = pub_v;
5  } else {
6      ms = set_msf(!b, ms);
7  }
8  x = p[5];
```

(a) write public data

```
1  ms = init_ms();
2  if b {
3      ms = set_msf(b, ms);
4      s[3] = sec_v;
5  } else {
6      ms = set_msf(!b, ms);
7  }
8  x = p[5];
```

(b) speculatively safe write

Figure 6: SLH protection not needed

Now consider the program from Figure 6b. Speculative execution of the store instruction at line 4 is *safe*, as the size of s is assumed to be greater than 3 and this store will not be speculated as the index is constant. Thus the address p[5] is unaffected, and the assignment at line 8 does not leak secret. Hence, x does not need to be protected.

4. Type System

In this section, we introduce a constraint-based type system which enforces speculative constant-time. We first present the type system and the typing rules, and state the soundness. Formal proofs are deferred to the appendix.

4.1. Security types

We consider the two-element security lattice $\{\mathsf{H}, \mathsf{L}\}$, with order $\mathsf{L} \leq \mathsf{H}$; as usual, we use the high-security level (H) for secret data and the low-security level (L) for public data. Other choices of lattices are possible, but not considered in this work. Formally, security levels are defined by the following grammar:

$$\tau := \ell \mid \mathsf{H} \mid \mathsf{L}$$

where ℓ ranges over level (or type) variables.

A security type is a pair $\tau = (\tau_n, \tau_s)$ of security levels. Intuitively, τ_n represents the security level under normal executions and τ_s represents the security level under all executions, including misspeculated ones. Throughout this paper, we require that the inequality $\tau_n \leq \tau_s$ hold, which justifies the following terminology:

- (L, L) denotes public data;
- (H, H) denotes secret data;
- (L, H) denotes transient data, i.e. data that is public under sequential execution but may depend on secrets under speculative execution.

In the remainder of the paper, we adopt the following conventions: we let τ_n and τ_s represent the first and second components respectively of a pair τ of security levels. Finally, we write $\tau \leq \tau'$ as shorthand for $\{\tau_n \leq \tau'_n, \tau_s \leq \tau'_s\}$.

4.2. Constraint sets

A constraint set is a set of inequalities of the form $\ell_1 \leq \ell_2$ where ℓ_1 and ℓ_2 are security levels.

The closure $\overline{C}$ of a constraint set C is the smallest set of constraints that contains C and is closed under transitivity. A

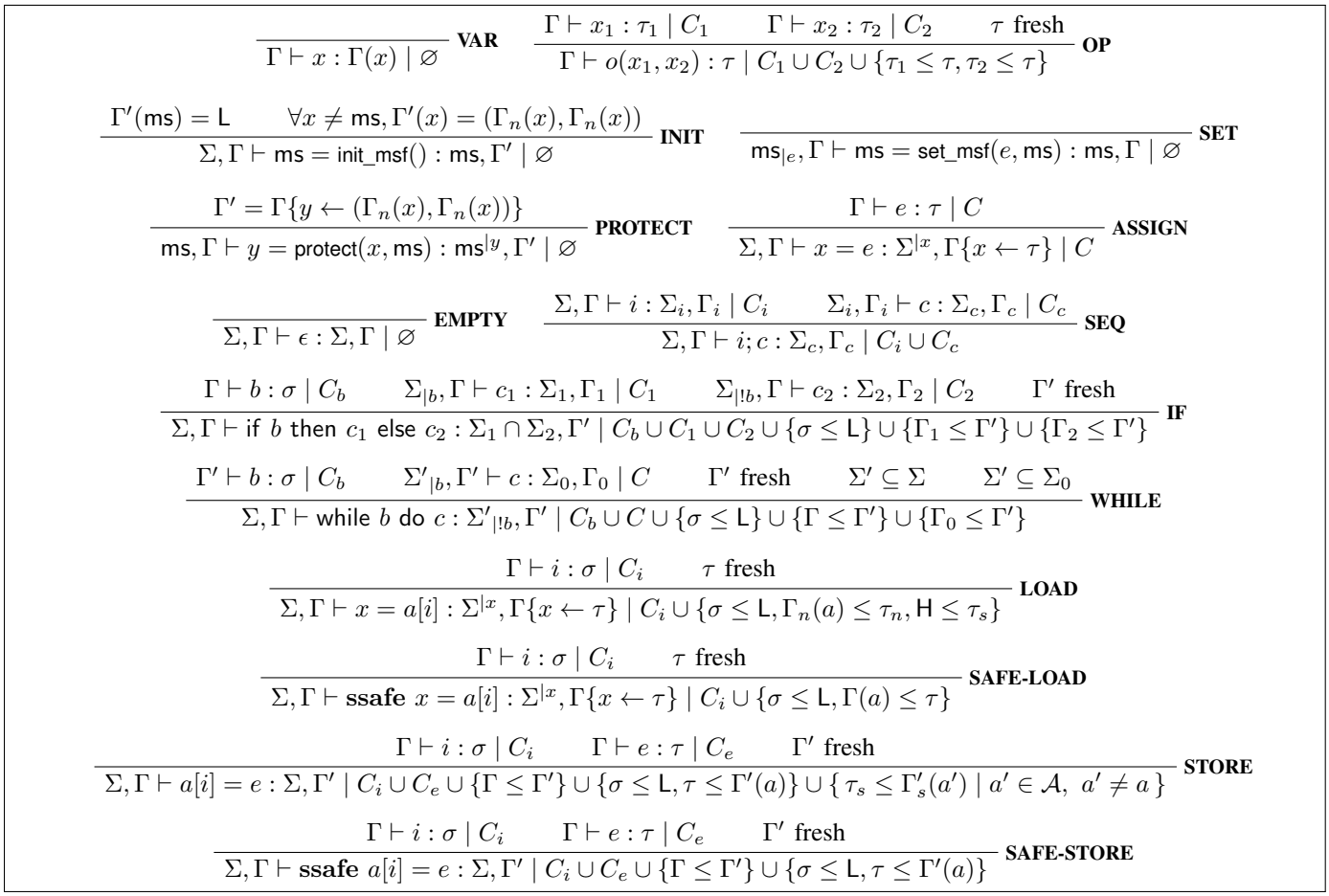


Figure 7: Typing rules

constraint set C is consistent if $(\mathsf{H} \leq \mathsf{L}) \notin \overline{C}$. A constraint set is consistent iff there exists a valuation θ mapping level variables to $\{\mathsf{L}, \mathsf{H}\}$ such that for every constraint $(\ell_1 \leq \ell_2) \in C$, $\theta(\ell_1) = \theta(\ell_2)$ or $\theta(\ell_1) = \mathsf{L}$ and $\theta(\ell_2) = \mathsf{H}$.

4.3. Misspecultion type (MSF-type)

Our type system assigns different types depending on whether execution is misspeculating or not. In order to capture this (specialized) form of value sensitivity, our type system uses MSF-type. The set of MSF-type is defined by the following grammar:

$$\Sigma = \text{unknown} \mid \mathsf{ms} \mid \mathsf{ms}_{|e}$$

The informal meaning of MSF-type is as follows:

- the type unknown states that we do not know if we are misspeculating or not, more precisely, that no register variable is expected to contain this information;
- the type ms states that if execution is misspeculating, then the register ms is set to -1. The type ms is used to make sure that values are correctly masked;
- the type $\mathsf{ms}_{|e}$ states that if execution is misspeculating, and the boolean expression e evaluates to true, then the register ms is set to -1. The type $\mathsf{ms}_{|e}$ is used by the type system to postpone the update of ms after a branching instruction.

In the remainder of the paper, we use the following notation:

$$\begin{array}{lcl} \Sigma_1 \subseteq \Sigma_2 & := & \Sigma_1 = \text{unknown} \vee \Sigma_1 = \Sigma_2 \\ \Sigma_1 \cap \Sigma_2 & := & \text{if } \Sigma_1 = \Sigma_2 \text{ then } \Sigma_1 \text{ else unknown} \\ \Sigma^{|x} & := & \text{if } x \in \mathrm{fv}(\Sigma) \text{ then unknown else } \Sigma \\ \Sigma_{|b} & := & \text{if } \Sigma = \mathsf{ms} \text{ then } \mathsf{ms}_{|b} \text{ else unknown} \end{array}$$

where $fv(\Sigma_{|b}) = \Sigma \cup fv(b)$

4.4. Typing rules

The type system manipulates judgments of the form

$$\Sigma, \Gamma \vdash c : \Sigma', \Gamma' \mid C$$

where Γ and Γ' are security environments, Σ and Σ' are MSF-type, and C is a set of constraints.

Figure 7 provides the typing rules for the language introduced in Section 3. We use the following conventions: if $\Gamma(x) = (\tau_n, \tau_s)$, we often write $\Gamma_n(x)$ (resp. $\Gamma_s(x)$) for τ_n (resp. τ_s). Moreover we write L for the pair (L, L).

The typing rules **VAR** and **OP** are used for typing expressions. They use a simpler auxiliary judgment $\Gamma \vdash e : \tau \mid C$ which does not mention MSF-type. The rule **VAR** inspects the type from the context and does not generate any constraint. The rule **OP** for binary operators collects constraints from the first and second arguments and generates a new constraint that forces the security level of $o(x_1, x_2)$ to be the maximum of the security levels of x_1 and x_2.

The typing rules **INIT** and **SET** are used to type the init_msf and set_msf primitives. The rule **INIT** ensures that the MSF-type is set to ms after initialization. Moreover the security level of ms is set to (L, L)—since it contains the value 0—and the security level of all remaining variables is set to $(\Gamma_n(x), \Gamma_n(x))$—since transient values are committed upon initialization. The rule **SET** requires that the MSF-type before the update is $\mathsf{ms}_{|e}$, and ensures that the MSF-type after the update is ms.

The rule **PROTECT** requires that the MSF-type be ms, and ensures that the output type of y is $(\Gamma_n(x), \Gamma_n(x))$. This reflects the fact that sequential execution will copy the value of x to y, and a misspeculated execution will set y to a default value—we do not set the speculative type of y to L because we require that $\tau_n \leq \tau_s$. Note that in case $\mathsf{ms} = y$, the output MSF-type is set to unknown.

The rule **ASSIGN** enforces absence of direct flows. In case $x = \mathsf{ms}$ or x occurs in b the rule also sets the result MSF-type to unknown. Rules **EMPTY** and **SEQ** are straightforward; notice that **SEQ** accumulates the constraints.

The rule **IF** checks that the guard has security type (L, L) and that both branches can be typed with MSF-type $\Sigma_{|b}$ and $\Sigma_{|!b}$ respectively. The rule $(\Sigma_1 \cap \Sigma_2)$ ensures that the result MSF-type is unknown if both branches disagree on their result MSF-type, and that the output security environment is the maximum of Γ_1 and Γ_2. This is done by introducing a fresh security environment Γ' and adding constraints $\Gamma_1 \leq \Gamma'$ and $\Gamma_2 \leq \Gamma'$. The rule **WHILE** follows the same idea; notice that the constraints on Γ' ensure that Γ' is a fixed point.

The rule **LOAD** ensures that the array is accessed with an index of type L, L and assigns to the target register x a fresh output type τ. τ is subject to the constraint that $\Gamma_n(a) \leq \tau_n$ to prevent direct flows under sequential execution, and that $\mathsf{H} \leq \tau_s$ to prevent direct flows under speculative execution, where the instruction is potentially unsafe. The rule **SAFE-LOAD** considers the specific case where the load is speculatively safe, as instructed by the **ssafe** annotation, which needs to be checked independently. In this case the constraint $\mathsf{H} \leq \tau_s$ can be replaced by $\Gamma_s(a) \leq \tau_s$.

The rule **STORE** ensures that the array is accessed with an index of type L, L and that the stored expression e has type τ. Because the store can potentially be performed at an arbitrary address, the type system creates a fresh environment Γ' and adds some constraints. $\Gamma \leq \Gamma'$ ensures that types in Γ' are at least the ones in Γ[1]. $\tau \leq \Gamma'(a)$ ensures that the new type associated with a is the least level of the stored expression. Finally the constraint $\tau_s \leq \Gamma'_s(a')$ captures the fact that the speculative type of all other arrays should be at least the speculative type τ_s of the stored value due to a speculatively unsafe store. The rule **SAFE-STORE** relaxes these constraints, assuming that the store is speculatively safe. In this case, it is not required to lift the speculative type of all other arrays.

1. Creating fresh environments is not necessary to prove the soundness of the system. It is, however, the expected behavior in practice, as not doing so would have negative implications: assigning a variable that was previously required to be public with a secret value would fail.

4.5. Soundness

The type system is sound, i.e. it only accepts speculative constant-time programs. In this section, we give an informal statement of soundness. Details, including missing definitions and proof sketches, are found in Appendix B.

For soundness, we show that if p is safe and

$$\Sigma, \Gamma \vdash p : \Sigma', \Gamma' \mid C$$

then p is $=^{\Sigma}_{\Gamma,C}$-SCT. Informally, the partial equivalence relation $=^{\Sigma}_{\Gamma,C}$ is defined as setting in relation states which coincide on their public parts, as defined by necessity from C and Γ, and are furthermore compatible with the MSF type Σ.

The first step of the proof is to show that for every two executions

$$p, s_1 \xrightarrow[d]{o_1} p_1, s'_1$$
$$p, s_2 \xrightarrow[d]{o_2} p_2, s'_2$$

such that $s_1 =^{\Sigma}_{\Gamma,C} s_2$, we have $o_1 = o_2$ and $p_1 = p_2$. Moreover, we show that there exist Σ_0, Γ_0, C_0 such that $\Sigma_0, \Gamma_0 \vdash p_1 : \Sigma', \Gamma' \mid C_0$ and $s'_1 =^{\Sigma_0}_{\Gamma_0,C_0} s'_2$. Using the latter, we can prove by induction on the length of the executions that executions started in related states yield equal leakage, as required.

5. Implementation in Jasmin

We have implemented our approach on top of the latest (public) version of the Jasmin [14], [15] framework for high-assurance cryptography. Since this version contains several unpublished extensions that must be addressed by the type system, we start with a brief description of the framework and of these recent extensions.

5.1. The Jasmin framework

The Jasmin framework consists of three main components: the Jasmin language, which provides cryptographic engineers maximal expressivity and control through a combination of high-level constructs and low-level idioms; the (formally verified) Jasmin compiler which generates efficient assembly that respects programmers' intentions; and formal verification tools for safety, correctness, provable security, and side-channel protection. For our purposes, only the Jasmin language is relevant, so we focus on the language and its recent extensions.

The Jasmin language supports "assembly in the head", a verification-friendly programming style that mixes structured control flow and (sugared) assembly-level instructions. In order to provide programmers with maximal control over the generated code, the Jasmin language requires that every variable declaration come with an attribute **reg** or **stack** that determine if the variable will be stored in a register or on the stack, in which case they will be accessed using load and store instructions. Stack variables can have type *word* or *array*. A recent extension allows array variables to have another optional attribute: **ptr**, which indicates that the variable contains a pointer to an array. This attribute is combined with the previous ones, so **reg ptr** means that the variable is a pointer stored in a register while **stack ptr** is a pointer stored on the stack.

Another recent extension is non inlined functions. Previously, functions were automatically and systematically inlined. However such inlining is not appropriate for implementations with a non-trivial call-graph. It is worth noting that function calls commonly use **reg ptr** as arguments.

Finally, the Jasmin language was extended with a system call randombytes. Previously, randomness was simply passed as a parameter. However, this is not viable for implementations that may need an arbitrary amount of randomness or to implement widely used cryptographic APIs. A call to randombytes takes a **reg ptr** as argument (a pointer to an array of fixed length), and fills the array with random bytes. We assume that randombytes is safe.

5.2. Implementation details

We have enriched the Jasmin framework with an intra-procedural analysis that implements our type system. Below we briefly mention some of the key aspects.

Functions. In order to enable intra-procedural analysis, our implementation infers the security levels for function's inputs and outputs together with a security effect ℓ. This level is an upper bound for the speculative stores potentially performed by the function outside its expected memory scope. The security effect is used in the rule for function calls to update the speculative type of the local variables of the caller, in a similar fashion to what is done by the **STORE** rule. This is similar to the handling of effectful functions in information-flow type systems.

For efficiency and readability reasons, our implementation performs a simplification phase after inferring the type and constraint set of a function. For a very simple example (without security effects), the universally quantified type and security constraint $\tau_x \times \tau_y \to \tau_z \mid \{\tau_x \leq \tau_z, \tau_y \leq \tau_z\}$ is replaced by the equivalent but more readable universally quantified type and constraint set $\tau \times \tau \to \tau \mid \varnothing$.

Pointer variables. The implementation associates two types, i.e. two pairs of security levels, to **ptr** variables. The first type is for the pointer while the second type is for the pointed data. The use of two types can be exploited by programmers when spilling pointers into stack memory. Again, having two security types for pointers is common in information-flow type systems for pointer languages.

System calls. The type system requires that the pointer and the length argument of randombytes be public L, and considers that the output is secret H. The type system assumes that the stack effect of the system call is high, which has the effect of setting to secret H the speculative types of all arrays—similar to a speculatively unsafe store.

Declassification. The type system implements a *declassify* construct that can be used by the programmer to declare intended leakage. Although our guarantees only hold for programs without declassification, We think that the techniques of [8] apply to this setting and yield a proof of relative non-interference for typable programs.

Constraint generation. Recall that a constraint is an inequality $l_1 \leq l_2$ between security levels. So a set of constraints can be represented with a directed graph where there is an edge between l_1 and l_2 iff $l_1 \leq l_2$.

Constraint generation can be implemented efficiently using a union-find data structure. Each time a constraint is added, the constraint generation algorithm checks if it creates a cycle. If a cycle is found, either it contains L and H and an error is immediately reported (providing a relatively good location error); or all the variables of the cycle are merged, allowing us to reduce the size of the graph.

5.3. Integration in Jasmin

We have extended the Jasmin language with our primitives, and security annotations to tag variables and arrays with security levels. Once written, programs are checked for safety using the Jasmin safety checker to guarantee that programs are safe. We could similarly extend the speculative safety checker from [17] to verify speculative safety annotations. However, our current implementation uses a naive (but generally sufficient) heuristics: all array accesses of the form $a[n]$ where n is a constant within the bounds of the array are considered annotated as speculatively safe.

Our implementation of the type system is typically used for checking speculative constant-time before compiling programs. Typing source programs generally simplifies analysis, but may occasionally cause a loss of precision due to the inability to verify side-conditions such as speculative safety or variable conditions on MSF-type. This is easily compensated by using compilation passes such as *inlining*, *loop unrolling* and *constant propagation* to reveal which array accesses have constant indices and can be marked as speculatively safe, or to make sure that information on MSF-type is not lost.

We also extend the lowering pass of the compiler to emit valid assembly code:

- ms = init_msf () is compiled into lfence; ms=0;
- ms = set_msf (e, ms) into a branchless conditional move ms = -1 if !e, where !e is the negation of e, and thus is not subject to speculation on x86 architectures;
- x = protect (x, ms) is compiled into x |= ms, where x is updated with bitwise OR operation between x and ms.

6. Application to crypto software

In this section we explain how we use the type system described in Section 5 to protect high-performance implementations of cryptographic primitives. We start with a detailed description of how we protect different implementations of the ChaCha20 [10] stream cipher to explain different techniques and performance trade-offs. We then briefly explain how we protect optimized implementations of Poly1305 [11], authenticated encryption using NaCl's "secretbox" operation built from XSalsa20 [18] and Poly1305, the X25519 elliptic-curve Diffie-Hellman key exchange [12] and finally the lattice-based key-encapsulation mechanism Kyber [13]. Protecting these primitives was part of a bigger effort of protecting a whole cryptographic library; we report more on the effort of protecting the full library in Section 7.

6.1. Protecting ChaCha20

The internal state of the ChaCha20 stream cipher is a 4×4 matrix of 32-bit values (64 bytes in total) that contains the secret key (32 bytes), the nonce (8 bytes or 12 bytes), a counter (8 or 4 bytes), and a 16-byte constant. The encryption of an input message m implementation goes as follows: 1) the internal state s is initialized with the secret key, nonce, counter, and public constant; 2) a copy of the state, k, is initialized; 3) k is transformed 10 times (while loop) by a $DoubleRound$ function (2 rounds each, 20 rounds in total); 4) each element of k is added (modulo 2^{32}) to the corresponding element of s; 5) the first 64 bytes of m are xor-ed with k. For subsequent blocks of m, the counter is incremented by one, and the described process starts over (while loops over the message length). If the length of m is not a multiple of 64, then k is computed as usual, and only the necessary keystream bytes from k are consumed. $DoubleRound$ consists of a sequence of additions (modulo 2^{32}), xors, and rotations, and it does not require any temporary values (registers).

To implement Chacha20 in Jasmin for AMD64 using standard instructions and registers, first, we need to consider that we only have 15 registers available – the stack pointer is reserved for the Jasmin compiler. Given that most of the CPU time is spent on $DoubleRound$ (which is called 10 times for every 64 bytes of keystream data) and that fully unrolling the loop is not viable as it would significantly increase the resulting machine-code size, this loop must be preserved as a loop in assembly, and there should be one free register at the time the loop's comparison execution to load the rounds counter. Considering this, $DoubleRound$ expects 14 values of k to be in registers at the entry and exit of this function ($k[14]$ and $k[15]$ are in the stack). Internally, $DoubleRound$ can use 1 more. The state s is in the stack. Ciphertext and plaintext pointers are also in the stack, along with the message length (used on an outer loop). These are loaded when needed, and sometimes, it is necessary to store elements of k in the stack to make up space for these values. Since the pointers and the message length are public and copied into the stack for performance reasons, they might need to be protected when loaded.

TABLE 1: Hardening ChaCha20 implementations

Version	SCT	A	I	L	F	P	U	M
Ref	×	3.40	435	103	0			
Ref one	×	3.47	457	114	0			
SLH	✓	3.86	551	114	1	75	12	0
Selective	✓	3.51	482	114	1	6	12	0
MMX/stack	✓	3.43	436	96	1	0	0	13
MMX/MMX	✓	3.56	436	84	1	0	0	37
Reordered	✓	3.40	438	104	1	0	0	0

For each variant, this shows whether it is secure (SCT), the asymptotic per-byte cost of the primitive (A; in CPU cycles), the numbers of assembly instructions (I), loads (L), fences (F), protect (P), set_msf (U), and MMX operations (M).

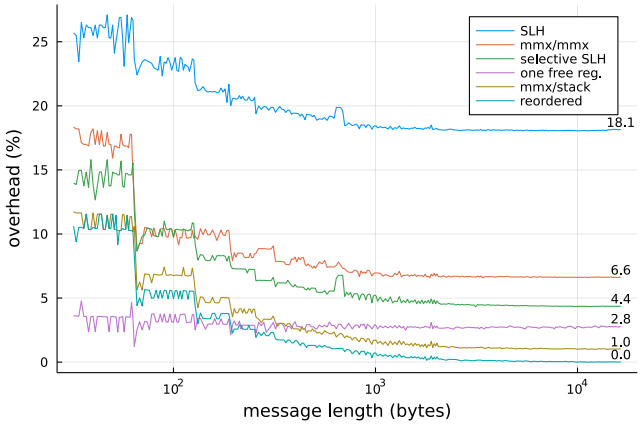


Figure 8: Run-time overhead for various implementations of ChaCha20

This section describes different implementations of ChaCha20, starting from a reference implementation and how a developer can protect against Spectre v1 using the three primitives introduced in Section 3. Figure 8 presents the overhead of different ChaCha20 versions compared to the original libjade's reference implementation, where all the available registers are used, and spills are minimized for maximum performance. Table 1 provides some metadata on the different implementations. Measurements have been performed on an Intel Core i7-8665U running NixOS 22.05 (Linux 5.15.58). Relative overhead is computed as the base-two logarithm of the ratio between execution times; the base of the logarithm is arbitrary and chosen so that a $100\,\%$ overhead means that execution time is twice as long.

Frequently, the first step to protect an implementation using SLH is to free one register to hold the misspeculation flag, which will be live during the entire program. The *one free register* implementation in Figure 8 corresponds to this first step, which shows an almost constant overhead (close to $2.8\,\%$) compared to the reference implementation. The overhead results from the fact that more moves into and from the stack are needed to keep one register free. Table 1 shows that the reference and *one free register* are not SCT.

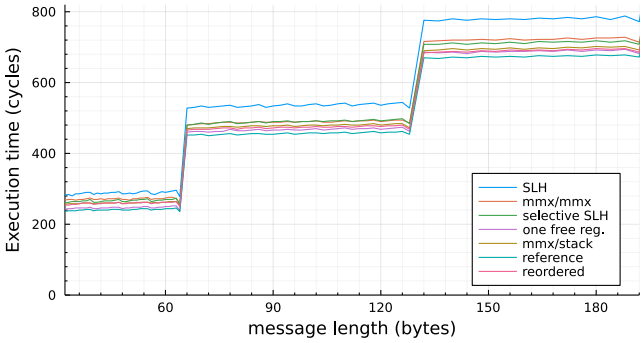


Figure 9: Performance of ChaCha20 implementations on small messages

SLH version corresponds to an implementation where SLH has been systematically applied: the code starts by initializing the misspeculation flag using init_msf, updates it after each branching instruction using set_msf, and every load is protected using protect. Notice that the role of initial init_msf is not only to initialize the MSF: it also guarantees that, initially, plain, output, and len do not speculatively depend on secret (H) data. While this version is accepted by the type checker and is secure, the overhead is significant (25 % for small messages, 18 % for large ones). This is not acceptable for implementations designed with high performance in mind. As for all the other protected versions, the highest performance overhead for small messages comes from the initial lfence. Figure 9 highlights that for small messages, although the relative overhead seems important (between 10 % and 25 %), the absolute overhead is of the order of a dozen cycles, which is arguably little.

Selective SLH is the version where SLH is applied selectively. The MSF is initialized and maintained as in the previous version, but only loaded data that can leak is protected. The type checker can drive the implementation's protection: an easy method is to run the type-checker iteratively, fix the issues it complains about, and continue the process until it stops complaining. This approach's overhead is more acceptable, less than 5 %, but we can do better.

Our first idea takes advantage of mmx registers: these were introduced in 1997 by Intel in the context of single instruction, multiple data (SIMD) instructions. In the reference implementation, three public variables must be spilled into the stack: plain, output, and len. $k[15]$ and $k[14]$ are also frequently moved into and from the stack. The first three contain public data (L), while $k[15]$ and $k[14]$ hold secret (H) data. The idea is to copy them into an mmx register instead of spilling them into the stack. Thus, they will not be subject to out-of-bound misspeculative store instructions. The main advantage of doing that is that no load (unspilling) needs to be protected, and the MSF is not needed anymore. This allows to free one register compared to the *one free register* implementation, and no set_msf instruction is required. Since k variables are H, they can remain on the stack. This leads to two new implementations: the first is *mmx/mmx* where the public variables and k variables are copied into mmx registers, and the second is *mmx/stack* where only the public variables are copied into mmx registers. The overhead for *mmx/stack* version is very low (1 % for large messages), which is better than what we obtain for the *selective SLH* version. For *mmx/mmx* the overhead is more significant, and the implementation is less efficient than the *selective SLH* version. This can be explained by the fact that $k[15]$ and $k[14]$ are intensively spilled/unspilled during the ChaCha20 computation. Moving a register to an mmx register and vice versa is typically more expensive than moving data between a register and the stack. The reason is that instructions moving data between integer and *mmx* registers are issued to execution units in the CPU rather than load/store units. As cryptographic code is typically bottlenecked by arithmetic rather than memory access, additional instructions going to execution units are more expensive than load/store instructions.

To conclude, we discuss the last protected version: *reordered*. The key idea of this version is to remark that the public variables need to be protected after unspilling because some potentially speculatively unsafe store can happen while these variables are on the stack (in this case, writing the computed ciphertext). By reordering the implementation's instructions for the unspill of public variables to happen before writing the output ciphertext, we achieve a negligible overhead ($< 0.3\%$) for large messages. In fact, this implementation can perform slightly better for sufficiently large messages due to the reordering causing a more efficient execution. There is still an overhead for small messages due to the lfence performed at the beginning of the function. It is worth mentioning that this technique can only be used in specific examples with suitable characteristics, such as the presented one.

6.2. Protecting Poly1305

Three Jasmin implementations of Poly1305, a message authentication code, are available: the reference implementation that uses standard AMD64 instructions and vectorized implementations, AVX, and AVX2. These implementations of libjade define two functions: 1) given a message m, the corresponding length $mlen$, and a 32-byte key k, a 16-byte authentication tag t is computed; 2) a function to allow the verification, in constant-time, of a given tag t for a given m, $mlen$, and the corresponding key k. Informally, and as an intuition, Poly1305 performs a series of multiplications (modulo $2^{130} - 5$) between the first half of k and each 16 bytes of m, while accumulating the result to be later added to the second half of k. Let $k = (r, s)$. In the context of the reference implementation, and considering that the accumulator has 130 bits and r has 128 bits, these can be represented using just three and two 64-bits registers. For this reason, this implementation does not need to perform any spill for either public or private data, given that it is possible to keep all variables on registers at all times. All 15 available registers are used. One init_msf instruction is enough to achieve Spectre v1 protection. AVX and AVX2 implementations use the reference implementation for small

messages (faster) and can also be protected with just one init_msf.

6.3. Protecting secretbox

The secretbox operation provides an interface for authenticated encryption. In this context, we protected three implementations of XSalsa20Poly1305: reference, AVX, and AVX2. XSalsa20 is similar to the previously discussed ChaCha20 motivating example. The main difference that is most relevant for the context of this discussion is that more spills are required because one temporary variable is needed during the rounds computation. These additional spills are done over secret data (k) and, as such, maintaining the misspeculation flag is unnecessary if mmx registers are used. One init_msf on each implementation is needed to protect the encryption in XSalsa20Poly1305, which produces a tag and the corresponding ciphertext. Briefly, the decryption occurs as follows: the authentication tag of the ciphertext is computed and compared with the tag given as an argument. If the computed and received tags are equal, then the decryption happens; otherwise, it does not, and the return value is set accordingly. Hence, the control flow depends on secret data: the result of verifying if the tag is valid or not. For this reason, the verification results must be declassified using declassify, and then, to prevent misspeculation on this value, one init_msf is used. The alternative would be to update the misspeculation flag and use protect. All XSalsa20Poly1305 implementations then require two init_msf and one declassify.

6.4. Protecting X25519

X25519 is, in its core, elliptic-curve scalar multiplication. It takes two inputs: a 32-byte scalar s and the x-coordinate of a point P on an elliptic curve, also encoded as a 32-byte string. Constant-time implementations use a fixed sequence of instructions to compute the x-coordinate of sP. This sequence of instructions is free of input-dependent memory access and free of input-dependent branches. As a consequence, the only modification required to add Spectre v1 protection is the fence (i.e., init_msf) right after entering the function.

While X25519 may not look like a very interesting case, it illustrates how the type system avoids enforcing unnecessary protect operations: Essentially all X25519 implementations, including ours, employ a fixed-length loop over the scalar bits to keep code-size within reasonable limits. The local loop-counter variable has to be public, so one might expect that spilling it to the stack requires a protect and that consequently we would need to add an set_msf to the branch. As state-of-the-art X25519 implementations targeting modern Intel CPUs typically do *not* use vector instructions, but work with standard 64-bit integer registers, register pressure is high and spilling the loop counter is very common practice. However, as the loop counter is spilled and retrieved *without any branch in between*, we can be sure that even in speculative execution we retrieve the (public) value we spilled before and that no protect is required.

6.5. Protecting Kyber

The high-speed implementation of Kyber is the most complex primitive in libjade in terms of lines of code and required protections. One reason is that Kyber uses multiple functions from the Keccak family [19] as subroutines. While the core of Kyber does not make heavy use of integer registers, the Keccak permutation does. This means that before calling into these routines we need spills of input and output pointers to free those registers. Retrieving those pointers later requires a protect and we need to use set_msf at each branch in between to update the misspeculation flag.

Another reason is that during key generation, encapsulation, and decapsulation, Kyber expands a public seed through SHAKE-128 and runs rejection sampling on the output to sample values that are uniform modulo $q = 3329$. In encapsulation the seed is provided as part of the public key and in decapsulation as part of the secret key. As values loaded from external memory (i.e., not jasmin-managed stack space) are considered secret, this requires declassifying the seed. Furthermore, the rejection-sampling loop requires set_msf and the actual rejection decision requires a protect.

7. Benchmarking and results

In this section we evaluate the cost of protecting cryptographic software against Spectre v1. We first evaluate cost in terms of developer effort, then consider the cost of type-checking Jasmin software with our protections, and finally measure cost in terms of computational overhead. As a starting point for all these cost metrics we consider code written in Jasmin that is already protected against traditional timing attacks.

More specifically, we target libjade, a cryptographic library that is currently in development and that is written entirely in Jasmin. It implements multiple symmetric primitives (hash functions, stream ciphers, and authenticators); elliptic-curve scalar multiplication using X25519 [12]; and lattice-based key encapsulation using Kyber [13].

The original libjade library[2] consists of about 16 k lines of Jasmin code and provides 72 entry points. For most primitives it has multiple implementations, including reference implementations and optimized implementations using different instruction-set extensions (e.g., the AVX2 vector-instruction extension). At the moment, Jasmin only supports AMD64 as target architecture, so all implementations are targeting this architecture.

7.1. Developer effort

We have adapted the complete library so that each entry point is protected against Spectre v1. In total this required inserting 79 init_msf, 74 set_msf, and 73 protect primitives. Insertion of init_msf is rather straight-forward: we need one such primitive for each entry point before performing any

2. Publicly available on the web: https://github.com/formosa-crypto/libjade/tree/ece99a3bbd8ebd831f285da0c909daba1ce2972d.

leaking operation (e.g., a load, store or branch) on any data. Additionally we need one init_msf after each call to randombytes; libjade has a total of 4 such calls, all in the implementations of Kyber. The remaining 3 init_msf primitives are in 3 different implementations of secretbox; as explained in Section 6 we choose to not track the misspeculation flag up to the declassification, but place an init_msf right after declassify instead. Insertion of set_msf and protect is not quite as straight-forward, but guided by the type system and corresponding compiler errors. Overall the developer effort is remarkably low. We emphasize that this efficiency is made possible by the enhanced type system in Jasmin. Without the help of this type system, insertion of protections would be much more cumbersome and, more importantly, error-prone.

7.2. Performance of the type-checker

Type-checking for SCT all implementations from the whole libjade library takes a total of a few seconds on a developer's laptop. For reference, the complete compilation of libjade from Jasmin to assembly takes a couple of minutes on the same laptop.

7.3. Computational overhead

To assess the run-time impact of our Spectre v1 protections added to libjade, we have measured the execution time of each primitive; for primitives with variable-length input we consider a sample of representative message sizes. All measurements were performed on a single core of a machine featuring an Intel Core i7-10700K (Comet Lake) CPU with hyperthreading and TurboBoost turned off. Each reported cycle count is the median of 8192 runs for primitives with fixed input length and of 1024 runs for each input length for primitives with variable input length. We followed the standard practice of carrying out the benchmarks on an otherwise idle machine. While this helps to reduce variance in cycle counts, it also means that the cost of fence instructions is measured on the optimistic side and may be larger on systems under full load.

Table 2 reports the measurements for the fastest implementation of each of the primitives we considered also in Section 6. We focus on the fastest implementation of each primitive here because this is the most relevant number for performance-critical applications. More extensive benchmark results (i.e., covering more primitives and more implementations of the same primitives) are generated by the artifact included with this submission.

The first thing we notice in these benchmark results is that the relative overhead for symmetric primitives decreases dramatically with increasing message length. As explained already in Section 6.1, the reason is the constant overhead caused by the initial init_msf. We comment that this cost is more due to the call from unprotected C code into Jasmin code than from the actual application of selective SLH. It would in principle be possible to eliminate this cost by compiling the calling code with SLH protection and forwarding the misspeculation flag to Jasmin. However, this would require an extension to the C function-call ABI to standardize how this flag is passed across function calls.

TABLE 2: Benchmark results of the fastest implementations of select primitives in libjade without Spectre v1 protections ("constant-time", CT) and with Spectre v1 protections ("speculative constant-time", SCT) on an Intel Core i7-10700K (Comet Lake) CPU

Primitive	Impl.	Op.	CT	SCT	overhead [%]
ChaCha20	avx2	32 B	314	352	12.10
	avx2	32 B xor	314	352	12.10
	avx2	128 B	330	370	12.12
	avx2	128 B xor	338	374	10.65
	avx2	1 KiB	1190	1234	3.70
	avx2	1 KiB xor	1198	1242	3.67
	avx2	1 KiB	18872	18912	0.21
	avx2	16 KiB xor	18970	18994	0.13
Poly1305	avx2	32 B	46	78	69.57
	avx2	32 B verif	48	84	75.00
	avx2	128 B	136	172	26.47
	avx2	128 B verif	140	170	21.43
	avx2	1 KiB	656	686	4.57
	avx2	1 KiB verif	654	686	4.89
	avx2	16 KiB	8420	8450	0.36
	avx2	16 KiB verif	8416	8466	0.59
secretbox	avx2	32 B	1104	1138	3.08
	avx2	32 B open	1862	1950	4.73
	avx2	128 B	1198	1234	3.01
	avx2	128 B open	1960	2044	4.29
	avx2	1 KiB	3066	3110	1.44
	avx2	1 KiB open	3886	3950	1.65
	avx2	16 KiB	31298	31376	0.25
	avx2	16 KiB open	32146	32208	0.19
X25519	mulx	smult	98352	98256	-0.098
	mulx	base	98354	98262	-0.094
Kyber512	avx2	keypair	25694	25912	0.848
	avx2	enc	35186	35464	0.790
	avx2	dec	27684	27976	1.055
Kyber768	avx2	keypair	42768	42888	0.281
	avx2	enc	54518	54818	0.550
	avx2	dec	43824	44152	0.748

The central result though is that for sufficiently long cryptographic computations in Jasmin (i.e., when the constant overhead of the init_msf becomes negligible), the performance impact is extremely low; typically less than 1%. The benchmark numbers that require explanation is the slightly negative overhead for X25519. We started investigating the reasons for these numbers and preliminary results suggest that they are due to different code alignment. We will continue to look into this.

8. Limitations

Our approach has two main limitations: it does not support declassification, and it operates on source Jasmin programs. We discuss both limitations, and then turn to more general concerns.

Declassification. Security-typed languages typically feature a declassification mechanism which allows programmers to

declare intended leaks of information. These leaks correspond to values that depend on secrets (for instance, ciphertexts) but can safely be viewed as public (as these values do not carry any information to parties without access to the keys). We have chosen not to consider declassification into our formal treatment. It is plausible that our results extend to declassification. However, accommodating such an extension would entail modifying the operational semantics of the language and the security policy (switching to relative notions as done for instance in [8]), and overall affect the readability of the paper.

Preservation. It is always delicate to verify security properties at source level, since compilers may break these properties. In the case of the Jasmin compiler, we know that all its passes are proved for their functional correctness and support preservation of constant-time [20] in Coq. The proof to carry out the preservation of constant-time property is done without any need to change the proofs of functional correctness. However, proving preservation of speculative constant-time poses some challenges. First, the semantics would need to be adapted even in order to define speculative constant-time. Second, it is not clear whether all passes of the Jasmin compiler will preserve speculative constant-time. One potential issue is for public arrays. In Jasmin, public arrays are lowered to memory accesses and this will create a security type mismatch and might break the SCT property. However, such arrays are rare in cryptographic code (and do not appear in the examples of this paper). We leave the issue for future work.

Other concerns. Protecting cryptographic code against Spectre v1 is necessary, but not sufficient to eliminate leakage of cryptographic secrets through speculative execution. One reason is that cryptographic libraries typically live in the same address space as application code, which means that Spectre gadgets in that code can also leak secrets. One way to address this issue is to also protect the application code against speculative attacks. Another option is to isolate the cryptographic code, for example, by running it in a separate "crypto-agent" process.

Furthermore, Spectre v1 is not the only transient execution attack vector beyond traditional timing attacks. Some of these additional attack vectors—most notably Meltdown, aka Spectre v3 [5]—are enabled by hardware or microcode bugs and eliminated by those bugs being fixed. Others, like Spectre v2, v4, and speculative attacks related to function returns [21], [22] are due to CPU *features* and will require dedicated mitigations. We note that most cryptographic software, including all code presented in this paper, does not use any indirect branches. This inherently protects against Spectre v2. For Spectre v4, the cheapest mitigation for Intel CPUs that we are aware of is to disable speculative store bypass [23].

9. Related work

There is a large body of work on enforcing and mitigating Spectre attacks. Following [24], this work can be classified according to its target policy. Typically, the target policy is some variant of speculative constant-time (SCT) [9], or relative constant-time (RCT) [25], a weaker property which ensures that speculative execution does not leak more than sequential execution—note that RCT is also called speculative non-interference in the literature.

Spectector [25] and Pitchfork [9] use symbolic execution to enforce RCT and SCT for programs with fences. Their symbolic semantics over-approximates the behavior of programs and cannot be used to verify programs that use Speculative Load Hardening. BinsecRel [26] uses relational symbolic execution to enforce SCT for programs with fences and index masking. In principle, the relational symbolic execution of BinsecRel is sufficiently precise to verify programs that are protected by Speculative Load Hardening. However, BinSecRel does not support all the language features required for high-speed cryptography.

Blade [16] is an automated tool that enforces SCT using fence and index masking—the latter can be more efficient than fencing, but requires that the size of arrays is known statically, which excludes algorithms that take arbitrary-length inputs as parameters. Blade is sound, i.e. it transforms every program that is typable with a constant-time system is transformed into a speculative constant-time program. The salient feature of Blade is that it carefully minimizes the number of protections. Precisely, Blade constructs a data-flow graph where nodes can be annotated as sources, i.e. they create a transient value, or sinks, i.e. they use a transient value in a leaking instruction. Blade then uses a classic min-cut algorithm to ensure that every path from sources to sinks is protected. However, the protected programs obtained by Blade are less efficient than programs based on Selective Speculative Load Hardening.

[17] defines an information-flow type system to protect Jasmin programs against Spectre attacks. Their type system enforces a stronger form of SCT that covers against v1 and a very limited form of v4 attacks. However, their approach is based on fences, which imposes a high performance overhead, and requires programs to be speculatively safe, which also incurs some performance overhead. In contrast, our type system is significantly more elaborate and accepts programs that are protected using (Selective) Speculative Load Hardening. Finally, our type system is implemented for the latest version of Jasmin, which includes many new challenging features.

[8] formalizes Selective Speculative Load Hardening. They also define a constant-time type system for a core language with fences and a declassify construct, and show that selSLH transforms typable programs into programs that satisfy RCT—it also follows from their results that typable programs without declassify are transformed into programs that satisfy SCT. In contrast, we offer mechanisms that can implement but are not limited to selSLH, and we prove that typable programs satisfy SCT. In addition, [8] uses

Pitchfork to estimate the performance benefits of Selective Speculative Load Hardening on three examples: ChaCha, donna and Ed25519. Our implementations are better than their predicted overhead.

10. Conclusion

We have designed an approach for protecting cryptographic software against Spectre v1 attacks at extremely low cost both in terms of developer time and computational overhead. We have used our approach for protecting multiple cryptographic primitives, including a highly optimized implementation of Kyber. We hope that our work will be a starting point to upgrade the gold standard of constant-time cryptography, and will help deliver new post-quantum implementations that are not only protected against attacks by future large quantum computers, but also against the most common classes of speculative attacks.

On a more foundational side, it would be interesting to prove that the Jasmin compiler preserves our protections, i.e. that typable programs are compiled to speculative constant-time assembly code. We hope that recent techniques for proving preservation of constant-time [27], [28], [20] can serve as a starting point to achieve this goal.

Acknowledgments

This research was supported by Deutsche Forschungsgemeinschaft (DFG, German research Foundation) as part of the Excellence Strategy of the German Federal and State Governments – EXC 2092 CASA - 390781972; by the European Commission through the ERC Starting Grant 805031 (EPOQUE); by the German Federal Ministry of Education and Research (BMBF) in the course of the 6GEM research hub under grant number 16KISK038; by the *Agence Nationale de la Recherche* (French National Research Agency) as part of the France 2030 programme – ANR-22-PECY-0006. Author Peter Schwabe is a member of the advisory boards of Bitmark Inc., PQShield, Neutrality, and SciEngines.

References

[1] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao, and B. Parno, "Sok: Computer-aided cryptography," in *IEEE S&P*, 2021, pp. 777–795.

[2] J. K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "HACL*: A verified modern cryptographic library," in *ACM CCS*, 2017, pp. 1789–1806.

[3] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala, "Simple high-level code for cryptographic arithmetic - with proofs, without compromises," in *IEEE S&P*, 2019, pp. 1202–1219.

[4] 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, "Spectre attacks: Exploiting speculative execution," in *IEEE S&P*, 2019, pp. 1–19.

[5] 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, "Meltdown: Reading kernel memory from user space," in *USENIX Security*, 2018, pp. 973–990.

[6] G. Alagic, D. Apon, D. Cooper, Q. Dang, T. Dang, J. Kelsey, J. Lichtinger, C. Miller, D. Moody, R. Peralta, R. Perlner, A. Robinson, D. Smith-Tone, and Y.-K. Liu, "Status report on the third round of the NIST post-quantum cryptography standardization process," NIST, Tech. Rep. NISTIR 8413, 2022.

[7] C. Carruth, "Speculative load hardening – a Spectre variant #1 mitigation technique," LLVM documentation, https://llvm.org/docs/SpeculativeLoadHardening.html.

[8] B. A. Shivakumar, J. Barnes, G. Barthe, S. Cauligi, C. Chuengsatiansup, D. Genkin, S. O'Connell, P. Schwabe, R. Q. Sim, and Y. Yarom, "Spectre declassified: Reading from the right place at the wrong time," in *IEEE S&P*, 2023 (to appear).

[9] S. Cauligi, C. Disselkoen, K. von Gleissenthall, D. M. Tullsen, D. Stefan, T. Rezk, and G. Barthe, "Constant-time foundations for the new Spectre era," in *PLDI*, 2020, pp. 913–926.

[10] D. J. Bernstein, "ChaCha, a variant of Salsa20," in *Workshop record of SASC*, 2008, https://cr.yp.to/chacha/chacha-20080120.pdf.

[11] ——, "The Poly1305-AES message-authentication code," in *FSE*, 2005, pp. 32–49.

[12] ——, "Curve25519: new Diffie-Hellman speed records," in *PKC*, 2006, pp. 207–228.

[13] J. W. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, "CRYSTALS – Kyber: A CCA-secure module-lattice-based KEM," in *IEEE EuroS&P*, 2018, pp. 353–367.

[14] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P. Strub, "Jasmin: High-assurance and high-speed cryptography," in *CCS*, 2017, pp. 1807–1823.

[15] J. B. Almeida, M. Barbosa, G. Barthe, B. Grégoire, A. Koutsos, V. Laporte, T. Oliveira, and P.-Y. Strub, "The last mile: High-assurance and high-speed cryptographic implementations," in *IEEE S&P*, 2020, pp. 965–982.

[16] M. Vassena, C. Disselkoen, K. v. Gleissenthall, S. Cauligi, R. G. Kıcı, R. Jhala, D. Tullsen, and D. Stefan, "Automatically eliminating speculative leaks from cryptographic code with Blade," in *POPL*, 2021.

[17] G. Barthe, S. Cauligi, B. Grégoire, A. Koutsos, K. Liao, T. Oliveira, S. Priya, T. Rezk, and P. Schwabe, "High-assurance cryptography in the Spectre era," in *IEEE S&P*, 2021, pp. 1884–1901.

[18] D. J. Bernstein, "Extending the Salsa20 nonce," in *Workshop record of Symmetric Key Encryption Workshop*, 2011, https://cr.yp.to/snuffle/xsalsa-20081128.pdf.

[19] M. J. Dworkin, *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*, NIST Std. FIPS PUB 202, 2015.

[20] G. Barthe, B. Grégoire, V. Laporte, and S. Priya, "Structured leakage and applications to cryptographic constant-time and cost," in *ACM CCS*, 2021, pp. 462–476.

[21] G. Maisuradze and C. Rossow, "ret2spec: Speculative Execution Using Return Stack Buffers," in *ACM CCS*, 2018, pp. 2109–2122.

[22] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *WOOT*, 2018.

[23] Intel, "Speculative store bypass / CVE-2018-3639 / INTEL-SA-00115," 2018, https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/speculative-store-bypass.html.

[24] S. Cauligi, C. Disselkoen, D. Moghimi, G. Barthe, and D. Stefan, "SoK: Practical foundations for Spectre defenses," 2022.

[25] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "Spectector: Principled detection of speculative information flows," in *IEEE S&P*, 2020, pp. 1–19.

[26] L. Daniel, S. Bardin, and T. Rezk, "Hunting the haunter – efficient relational symbolic execution for Spectre with haunted RelSE," in *NDSS*, 2021.

[27] G. Barthe, B. Grégoire, and V. Laporte, "Secure compilation of side-channel countermeasures: The case of cryptographic "constant-time"," in *CSF*, 2018, pp. 328–343.

[28] G. Barthe, S. Blazy, B. Grégoire, R. Hutin, V. Laporte, D. Pichardie, and A. Trieu, "Formal verification of a constant-time preserving C compiler," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, pp. 7:1–7:30, 2020.

Appendix

1. Semantics

Thus far, we have used s, b to represent the state of a program, without making the register map and memory explicit. From now on, we make these explicit and consider a state s as a triplet $\langle \rho, \mu, b \rangle$ where the register map ρ assigns values to registers, and the memory map μ assigns values to valid addresses, i.e. pairs of array names and indices such that the index lies in the range of the array, and b is the misspeculation flag. Judgments are of the form $p, s \xrightarrow[d]{o} p', s'$ and state that a single step of execution of the program p, under the directive d, updates the state s to a state s' producing the observation o, with a remaining program p'. The rules are given in Figure 10. The rules are taken from [8], [17].

The **ASSIGN** rule is standard. Our language only manipulates computations on registers, which implies no leakage may occur. The same applies for the **SET** and **INIT** rules, which use an assign instruction at their core, with specific values assigned: **SET** updates the value ms based on the evaluation of the boolean expression e; **INIT** sets ms to 0 and only works for non-speculative executions (it acts like a fence operator), as indicated by the MSF set to $\bot$.

The **PROTECT** is similar to an **ASSIGN** one, with the difference that it works by OR-masking the operand with ms. Here this masking operation is simplified to an equality test to -1 for the mask, which is the absorbing element for the OR operator.

The **LOAD** and **STORE** rules have safe and unsafe versions. Note that in all four cases, the address is leaked. The behavior of the unsafe versions remains more controlled than in other approaches [17], as they are not considered to leak the whole stack memory μ. Because of the assumption that all given programs are safe, these unsafe executions can only happen under speculative execution ($b = \top$). This also means that speculatively-unsafe programs can be executed, and that these programs are not prevented to be speculative constant-time. In both unsafe rules, the attacker is also given the initiative to access any desired memory location.

Finally, the flow-control rules **COND** and **WHILE** have different behaviors based on the directive used. As a result, these rules are the only ones where the MSF is subject to change, which depends on whether the directive is force. The leakage consists of the *executed* branch.

Our proof of soundness assumes that all annotations of speculative safety are correct.

Definition 2 (Speculative safety)**.** *A* load *or* store *instruction of the form* $c(a, e) = x := a[e]$ *or* $c(a, e) = a[e] := x$ *is speculatively safe in a program-state configuration* p, s *if:*

$$\overline{\forall} \quad p, s \xrightarrow[D]{\ell} \textbf{ssafe}\ c(a,e); p', \langle \rho, \mu, b \rangle \implies [\![e]\!]_\rho \in |a| \,.$$

All **ssafe** *c annotations in the input program are assumed to verify this property.*

2. Soundness

We first introduce state indistinguishability. Recall that in a classic information-flow setting, two states are indistinguishable if they coincide their public variables. In our setting, the definition of state indistinguishability depends on the misspeculation flag. Informally, if the misspeculation flag is set to false, then two states are indistinguishable if their public variables coincide. In contrast, if the misspeculation flag is set to true, then two states are indistinguishable if their transient values coincide. In order to make this intuition formal, we parameterize the definition of state indistinguishability by a typing environment Γ which associates type variables to variables, and a valuation θ which associates a security level to each variable ℓ, and a speculative type Σ. The latter is used to determine if the misspeculation flag is set to true or not. Naturally, we need to restrict state indistinguishability w.r.t. Σ to pairs of states that are compatible with Σ, and in particular have their misspeculation flag to the same value.

Definition 3 (Indistinguishability relationship)**.** *For every* Γ, θ, Σ *and every* $\rho_1, \rho_2, \mu_1, \mu_2, b_1, b_2$*, the binary relationship* $=^{\Sigma}_{\theta\Gamma}$ *is defined by:*

$$\langle \rho_1, \mu_1, b_1 \rangle =^{\Sigma}_{\theta\Gamma} \langle \rho_2, \mu_2, b_2 \rangle \equiv \begin{cases} b_1 = b_2 \\ \rho_1 =_{\theta\Gamma_s} \rho_2 \\ \neg b_1 \implies \rho_1 =_{\theta\Gamma_n} \rho_2 \\ \mu_1 =_{\theta\Gamma_s} \mu_2 \\ \neg b_1 \implies \mu_1 =_{\theta\Gamma_n} \mu_2 \\ \rho_1, b_1 \vDash \Sigma \\ \rho_2, b_2 \vDash \Sigma \end{cases}$$

where:

$$x =_\tau y \equiv \tau = \mathsf{L} \implies x = y$$
$$\rho =_f \sigma \equiv \forall x \in \mathrm{dom}(\rho), \rho(x) =_{f(x)} \sigma(x)$$
$$\mu =_f \nu \equiv \forall (a, n) \in \mathrm{dom}(\mu),\ \mu(a, n) =_{f(a)} \nu(a, n)$$

and the MSF state correspondence is defined as follows:

$$\begin{array}{ll} \rho, b \vDash \bot & \textit{is always valid} \\ \rho, b \vDash \mathsf{ms} & \equiv b \iff [\![\mathsf{ms}]\!]_\rho = -1 \\ \rho, \top \vDash \mathsf{ms}_{|e} & \equiv [\![e]\!]_\rho = \top \implies [\![\mathsf{ms}]\!]_\rho = -1 \\ \rho, \bot \vDash \mathsf{ms}_{|e} & \equiv [\![e]\!]_\rho = \top \wedge [\![\mathsf{ms}]\!]_\rho \neq -1 \end{array}.$$

Given Σ, Γ, C*, we let* $=_{\Sigma,\Gamma,C}$ *denote the union of* $=^{\Sigma}_{\theta\Gamma}$ *over all* θ *such that* $\theta \models C$*.*

We establish several simple lemmas which we use in the soundness proofs.

$$\frac{}{S \xrightarrow[\epsilon]{\epsilon} S}\ \textbf{0-STEP} \qquad \frac{S \xrightarrow[d]{o} S' \qquad S' \xrightarrow[D]{O} S''}{S \xrightarrow[d::D]{o::O} S''}\ \textbf{S-STEP} \qquad \frac{}{x := e; c,\ \langle \rho, \mu, b\rangle \xrightarrow[\mathsf{step}]{\bullet} c,\ \langle \rho\{x \leftarrow [\![e]\!]_\rho\}, \mu, b\rangle}\ \textbf{ASSIGN}$$

$$\frac{[\![e]\!]_\rho = n \in |a| \qquad \mu(a,n) = v}{x := a[e]; c,\ \langle \rho, \mu, b\rangle \xrightarrow[\mathsf{step}]{\mathsf{addr}(n)} c,\ \langle \rho\{x \leftarrow v\}, \mu, b\rangle}\ \textbf{LOAD} \qquad \frac{[\![e]\!]_\rho = n \in |a|}{a[e] := e'; c,\ \langle \rho, \mu, b\rangle \xrightarrow[\mathsf{step}]{\mathsf{addr}(n)} c,\ \langle \rho, \mu\{(a,n) \leftarrow [\![e']\!]_\rho\}, b\rangle}\ \textbf{STORE}$$

$$\frac{[\![e]\!]_\rho = n \notin |a| \qquad m \in |\alpha|\,, \mu(\alpha, m) = v}{x := a[e]; c,\ \langle \rho, \mu, \top\rangle \xrightarrow[\mathsf{load}(\alpha,m)]{\mathsf{addr}(n)} c,\ \langle \rho\{x \leftarrow v\}, \mu, \top\rangle}\ \textbf{U-LOAD} \qquad \frac{[\![e]\!]_\rho = n \notin |a| \qquad \alpha \in \mathcal{A}, m \in |\alpha|}{a[e] := e'; c,\ \langle \rho, \mu, \top\rangle \xrightarrow[\mathsf{store}(\alpha,m)]{\mathsf{addr}(n)} c,\ \langle \rho, \mu\{(\alpha,m) \leftarrow [\![e']\!]_\rho\}, \top\rangle}\ \textbf{U-ST}$$

$$\frac{m = \text{if } ([\![e]\!]_\rho = \top) \text{ then } [\![\mathsf{ms}]\!]_\rho \text{ else } -1}{\mathsf{ms} = \mathsf{set_msf}(e, \mathsf{ms}); c,\ \langle \rho, \mu, b\rangle \xrightarrow[\mathsf{step}]{\bullet} c,\ \langle \rho\{\mathsf{ms} \leftarrow m\}, \mu, b\rangle}\ \textbf{SET} \qquad \frac{\rho' = \rho\{\mathsf{ms} \leftarrow 0\}}{\mathsf{ms} = \mathsf{init_msf}(); c,\ \langle \rho, \mu, \bot\rangle \xrightarrow[\mathsf{step}]{\bullet} c,\ \langle \rho', \mu, \bot\rangle}\ \textbf{INIT}$$

$$\frac{v = \text{if } ([\![\mathsf{ms}]\!]_\rho = -1) \text{ then } -1 \text{ else } [\![x]\!]_\rho}{y = \mathsf{protect}(x, \mathsf{ms}); c,\ \langle \rho, \mu, b\rangle \xrightarrow[\mathsf{step}]{\bullet} c,\ \langle \rho\{y \leftarrow v\}, \mu, b\rangle}\ \textbf{PROTECT}$$

$$\frac{e = \text{if } (d = \mathsf{force}) \text{ then } \neg[\![t]\!]_\rho \text{ else } [\![t]\!]_\rho}{\text{if } t \text{ then } c_\top \text{ else } c_\bot; c,\ \langle \rho, \mu, b\rangle \xrightarrow[d]{\mathsf{branch}(e)} c_e; c,\ \langle \rho, \mu, b \vee d = \mathsf{force}\rangle}\ \textbf{COND}$$

$$\frac{e = \text{if } (d = \mathsf{force}) \text{ then } \neg[\![t]\!]_\rho \text{ else } [\![t]\!]_\rho \qquad c_\top = c_0; \text{while } t \text{ do } c_0; c \qquad c_\bot = c}{\text{while } t \text{ do } c_0; c,\ \langle \rho, \mu, b\rangle \xrightarrow[d]{\mathsf{branch}(e)} c_e,\ \langle \rho, \mu, b \vee d = \mathsf{force}\rangle}\ \textbf{WHILE}$$

Figure 10: Operational semantics

Lemma 1. $\overline{\forall}\quad f \leq g \wedge \rho =_f \sigma \implies \rho =_g \sigma$, *where* ρ *and* σ *are register memory functions or stack memory functions.*

Proof. There are only three possibilities for the function argument, and all of them yield the result trivially. □

The lemma entails that two indistinguishable states with respect to a certain level of privacy remain indistinguishable with respect to a level of privacy where more variables are considered high:

Lemma 2 (Lifting lemma)**.**

$$\overline{\forall}\quad f \leq g \wedge \Sigma' \subseteq \Sigma \wedge s_1 =_f^{\Sigma} s_2 \implies s_1 =_g^{\Sigma'} s_2,$$

where s_1 *and* s_2 *are program states and* f *and* g *functions returning pairs of types.*

We now turn to soundness.

Theorem 1 (Soundness)**.** *If* p *is safe and* $\Sigma, \Gamma \vdash p : \Sigma', \Gamma' \mid C$*, then for all valuations* θ *such that* $\theta \vDash C$*,* p *is* $=_{\theta\Gamma}^{\Sigma}$*-SCT.*

The remaining of this section is devoted to a proof of the theorem. Note that for convenience of reasoning, notably for dealing with conditionals and loops, we consider an augmented type system with a **WEAK** rule:

$$\frac{\Sigma_0, \Gamma_0 \vdash c : \Sigma_0', \Gamma_0' \mid C \qquad \Sigma_0 \subseteq \Sigma, \Sigma' \subseteq \Sigma_0'}{\Sigma, \Gamma \vdash c : \Sigma', \Gamma' \mid C \cup \{\Gamma \leq \Gamma'\} \cup \{\Gamma_0' \leq \Gamma_0\}}\ \textbf{WEAK}$$

We first prove soundness for expressions.

Lemma 3 (Soundness for expressions)**.**

$$\overline{\forall}\quad \Gamma \vdash e : (\tau_n, \tau_s) \mid C \wedge \theta \vDash C \wedge \rho_1 =_{\theta\Gamma_i} \rho_2 \implies [\![e]\!]_{\rho_1} =_{\theta(\tau_i)} [\![e]\!]_{\rho_2}, \textit{where} i = n \vee i = s.$$

Proof. By induction on the expression e. □

Next, we prove prove soundness for one-step execution. Note that our proof establishes additional conclusions, which correspond to premises which will fit appropriately in a proof by induction for multi-step executions.

Theorem 2 (One-step soundness)**.**

$$\begin{aligned}\overline{\forall}\quad & p, s_1 \xrightarrow[d]{o_1} p_1, s_1' \wedge p, s_2 \xrightarrow[d]{o_2} p_2, s_2' \wedge \\ & \Sigma, \Gamma \vdash p : \Sigma^*, \Gamma^* \mid C \wedge \theta \vDash C \wedge s_1 =_{\theta\Gamma}^{\Sigma} s_2 \\ & \implies o_1 = o_2 \wedge p_1 = p_2 \\ & \wedge (\exists \Sigma_0, \Gamma_0, C_0 \quad \Sigma_0, \Gamma_0 \vdash p_1 : \Sigma^*, \Gamma^* \mid C_0 \\ & \wedge s_1' =_{\theta\Gamma_0}^{\Sigma_0} s_2' \wedge \theta \vDash C_0)\end{aligned}$$

Proof. Let $p, \Gamma, \Gamma^*, s_1, s_2, \ldots$ be all variables universally bound in the expression, satisfying the hypotheses. More precisely, $s_i = \langle \rho_i, \mu_i, b\rangle$ and $s_i' = \langle \rho_i', \mu_i', b'\rangle$ (for $i = 1, 2$). Because the program p has a one-step execution, it cannot be ϵ. It is thus of the form $p = i; p'$. This means the typing derivation either ends with a **WEAK** or a **SEQ**. In the first case, we shall proceed by induction on the typing judgment. It is of the form

$$\frac{\Sigma', \Gamma' \vdash p : \Sigma^{*\prime}, \Gamma^{*\prime} \mid C'' \qquad \Sigma' \subseteq \Sigma, \Sigma^* \subseteq \Sigma^{*\prime}}{\Sigma, \Gamma \vdash p : \Sigma^*, \Gamma^* \mid C'' \cup \{\Gamma^{*\prime} \leq \Gamma^*\} \cup \{\Gamma \leq \Gamma'\}} \textbf{ WEAK}$$

However, by assumption, $\theta \vDash \{\Gamma \leq \Gamma'\}$, which entails $\theta\Gamma \leq \theta\Gamma'$. This implies, in addition to $\Sigma' \subseteq \Sigma$ via Lemma 2, that $s_1 =^{\Sigma'}_{\theta\Gamma'} s_2$. Besides, $\theta \vDash C''$. One can now apply the induction hypothesis and obtain corresponding values Σ_0, Γ_0, C_0, with $\theta \vDash C_0$. This allows us to write:

$$\frac{\Sigma_0, \Gamma_0 \vdash p_1 : \Sigma^{*\prime}, \Gamma^{*\prime} \mid C_0 \qquad \Sigma^* \subseteq \Sigma^{*\prime}}{\Sigma_0, \Gamma_0 \vdash p_1 : \Sigma^*, \Gamma^* \mid C_0 \cup \{\Gamma^{*\prime} \leq \Gamma^*\}} \textbf{ WEAK}$$

with $\theta \vDash C_0 \cup \{\Gamma^{*\prime} \leq \Gamma^*\}$ by assumption. The rest of the conclusions follow, which finishes the case.

Else, the last rule is a **SEQ**:

$$\frac{\Sigma, \Gamma \vdash i : \Sigma', \Gamma' \mid C' \qquad \Sigma', \Gamma' \vdash p' : \Sigma^*, \Gamma^* \mid C_p}{\Sigma, \Gamma \vdash i; p' : \Sigma^*, \Gamma^* \mid C' \cup C_p} \textbf{ SEQ}$$

and c is an instruction. We now proceed by induction on the typing derivation of i. This induction only has one inductive case, that of the **WEAK** rule. In that case, we have:

$$\frac{\Sigma_i, \Gamma_i \vdash i : \Sigma'_i, \Gamma'_i \mid C'_i \qquad \Sigma_i \subseteq \Sigma, \Sigma' \subseteq \Sigma'_i}{\Sigma, \Gamma \vdash i : \Sigma', \Gamma' \mid C'_i \cup \{\Gamma \leq \Gamma_i\} \cup \{\Gamma'_i \leq \Gamma'\}} \textbf{ WEAK}$$

which can then be used to produce a new, shorter, typing derivation for $i; p'$, as shown in Figure 11. It is now necessary to verify the other premises of the induction hypothesis, in order to apply it. We know that $\theta \vDash C'_i \cup C_p \cup \{\Gamma'_i \leq \Gamma'\}$. Besides $\theta \vDash \{\Gamma \leq \Gamma_i\}$, which entails $\theta\Gamma \leq \theta\Gamma_i$. Hence, Lemma 2 ensures that, also knowing that $\Sigma_i \subseteq \Sigma$, $s_1 =^{\Sigma_i}_{\theta\Gamma_i} s_2$, and we now apply the induction hypothesis with a shorter typing derivation for i, which finishes the case.

The remaining base cases will be discussed based on the semantic rules being applied (this rule is the same for both executions), which effectively filters which typing rule can be used with it. In all these cases but the last two, it is observed that $p'_i = p'$. Thus, the sought-after existential variables can be set to $\Sigma_0 = \Sigma'$, $\Gamma_0 = \Gamma'$ and $C_0 = C'$. And these cases will only consist in showing that $s'_1 =^{\Sigma'}_{\theta\Gamma'} s'_2$, and $o_1 = o_2$. It is easy to prove that misspeculation flags remain identical through both executions, and it will not be detailed further in the rest of the proof.

Unless stated otherwise, each case uses the same notations as the ones in the definition of the type system. Let us start the case discussion:

ASSIGN $i = x := e$. The typing rule is necessarily **ASSIGN**. For $y \neq x$, $\Gamma'_s(y) = \Gamma_s(y)$, and, for $i = 1, 2$, $\rho'_i(y) = \rho_i\{x \leftarrow [\![e]\!]_{\rho_i}\}(y) = \rho_i(y)$, but $\rho_1(y) =_{\theta(\Gamma_s(y))} \rho_2(y)$ by assumption; thus $\rho'_1(y) =_{\theta\Gamma'_s(y)} \rho'_2(y)$. As for x, $\Gamma'_s(x) = \tau$ and $\rho'_1(y) = [\![e]\!]_{\rho_1} =_\tau [\![e]\!]_{\rho_2} = \rho'_2(y)$, by lemma 3. Consequently, $\rho'_1 =_{\theta,\Gamma'_s} \rho'_2$. The same applies for Γ'_n, if $b = \bot$. Stack memory remains unchanged.
As for MSF, if $y \in \mathrm{fv}(\Sigma)$, then $\Sigma' = \bot$ and the condition is trivial. Else, it is easy to substitute ρ_i by ρ'_i $(i = 1, 2)$ in the MSF predicate, as all memory evaluations in it remain identical. Thus $s'_1 =^{\Sigma'}_{\theta\Gamma'} s'_2$. Leakage is trivially identical.

LOAD $i = x := a[i],\ d = \mathsf{step}$. The typing rule can correspond to either a **LOAD** or a **SAFE-LOAD** rule; this proof only uses constraints that are parts of both. Let σ be the type attributed to i. First, $\theta(\sigma_s) = \mathsf{L}$ thus $n = [\![i]\!]_{\rho_1} = [\![i]\!]_{\rho_2}$ by lemma 3, which entails $o_1 = o_2$. Just like in the previous case, for $y \neq x$, $\rho'_1(y) =_{\theta\Gamma'_s(y)} \rho'_2(y)$. If $\theta\Gamma'_s(x) = \mathsf{L}$, then $\theta\Gamma_s(a) = \mathsf{L}$ by constraint, however by assumption $\mu_1 =_{\theta\Gamma_s} \mu_2$; together with the previous type equality, this entails: $\forall n \in |a|,\ \mu_1(a, n) = \mu_2(a, n)$. Thus $\rho'_1(x) = \rho'_2(x)$. Consequently $\rho'_1(x) =_{\theta\Gamma'_s(x)} \rho'_2(x)$, and $\rho'_1 =_{\theta\Gamma'_s} \rho'_2$. The same applies for Γ'_n if $b = \bot$.
Regarding MSF, the reasoning is the same as in the previous case. Thus $s'_1 =^{\Sigma'}_{\theta\Gamma'} s'_2$.

U-LOAD $i = x := a[i],\ d = \mathsf{load}(\alpha, m)$. The typing rule can only correspond to a **LOAD**, for we assume all **ssafe** c annotations are correct (and an unsafe *load* would contradict this). As before, $n = [\![i]\!]_{\rho_1} = [\![i]\!]_{\rho_2}$ and $o_1 = o_2$. Just like in the previous case, for $y \neq x$, $\rho'_1(y) =_{\theta\Gamma'_s(y)} \rho'_2(y)$. Besides, $\theta\Gamma'_s(x) = \mathsf{H}$, thus $\rho'_1(x) =_{\theta\Gamma'_s(x)} \rho'_2(x)$. Hence, $\rho'_1 =_{\theta\Gamma'_s} \rho'_2$. Finally, there is no need to check for Γ_n, since necessarily $b = \top$. Indeed, one could otherwise derive from this execution a non-speculative execution with an out-of-bound addressing, which contradicts the assumption that all programs are *safe*. MSF works the same as before. Thus $s'_1 =_{\theta\Gamma'} s'_2$.

STORE $i = a[i] := e$. The typing rule is either **STORE** or **SAFE-STORE**. This case only uses constraints that are parts of both. Let τ be the type of e. As before, $n = [\![i]\!]_{\rho_1} = [\![i]\!]_{\rho_2}$ and $o_1 = o_2$. The constraints and the assumptions allow the use of lemma 1, which entails $\rho'_1 = \rho_1 =_{\theta\Gamma'_s} \rho_2 = \rho'_2$ (and the same holds for Γ'_n if needed) and $\mu_1 \doteq_{\theta\Gamma'_s} \mu_2$. As for the proof that $\mu'_1 =_{\theta\Gamma'_s} \mu'_2$, the result is also straightforward: For any array $b \neq a$, and any index $j \in |b|$, $\mu'_1(b, j) = \mu_1(b, j) =_{\theta\Gamma'_s} \mu_2(b, j) = \mu'_2(b, j)$. As for a, if $\theta\Gamma'_s(a) = \mathsf{L}$, then $\theta(\tau_s) = \mathsf{L}$ by constraint. Which means $\mu'_1(a, n) = [\![e]\!]_{\rho_1} = [\![e]\!]_{\rho_2} = \mu'_2(a, n)$. Hence for every $j \in |a|$, $\mu'_1(a, n) =_{\theta\Gamma'_s} \mu'_2(a, n)$. Finally, $\mu'_1 =_{\theta\Gamma_s} \mu'_2$. The proof of $\mu'_1 =_{\theta\Gamma'_s} \mu'_2$ is similar.
The MSF predicates remain unchanged: $\Sigma' = \Sigma$, and so do the register memories, which easily entails $\rho'_i, b \vDash \Sigma'$ $(i = 1, 2)$. This concludes this case: $s'_1 =^{\Sigma'}_{\theta\Gamma'} s'_2$.

U-STORE $i = a[i] := e, d = \mathsf{store}\ (b, j)$. This case has the same structure as the one before, using a different array name. It is worth noting that the set of additional constraints $\{\, \tau_s \leq \Gamma'_s(a') \mid a' \in \mathcal{A},\ a' \neq a \,\}$ contains the adequate constraint for b, but only its speculative type. Indeed, there is no need to prove $\mu'_1 =_{\theta\Gamma'_n} \mu'_2$ because, as in the case of **U-LOAD**, the speculative state is set $(b = \top)$.

INIT $i = \mathsf{ms} = \mathsf{init_msf}(),\ b = \bot$. Assumption gives $\rho'_1 = \rho_1 =_{\theta\Gamma_n} \rho_2 = \rho'_2$ then $\rho'_1 =_{\theta\Gamma'_s} \rho'_2$, and so on for all four equality. This first equality has to be further justified by the fact that, knowing that $\theta\Gamma'_s(\mathsf{ms}) = \mathsf{L}$: $0 = [\![\mathsf{ms}]\!]_{\rho'_1} =_\mathsf{L} [\![\mathsf{ms}]\!]_{\rho'_2} = 0$. The same applies to the register maps. The MSF predicate is as straightforward to verify. This finishes the case, $s'_1 =_{\theta\Gamma'} s'_2$.

SET $i = \mathsf{ms} = \mathsf{set_msf}(e, \mathsf{ms})$. Throughout the paragraph, $j = 1, 2$. We know $\rho_j, b \vDash \Sigma$. If $[\![e]\!]_{\rho_j} = \top$, then, $\rho'_j(\mathsf{ms}) = \rho_j(\mathsf{ms})$. Then, either $b = \top$, in which case the value of ms remains -1, which validates $\rho'_j, \top \vDash \mathsf{ms}$. Or $b = \bot$, and because of $\rho_j, \bot \vDash \mathsf{ms}_{|e}$, ms remains different from -1, which

$$\dfrac{\Sigma_i, \Gamma_i \vdash i : \Sigma_i', \Gamma_i' \mid C_i' \qquad \dfrac{\Sigma', \Gamma' \vdash p' : \Sigma^*, \Gamma^* \mid C_p \qquad \Sigma' \subseteq \Sigma_i'}{\Sigma_i', \Gamma_i' \vdash p' : \Sigma^*, \Gamma^* \mid C_p \cup \{\Gamma_i' \leq \Gamma'\}}\ \textbf{WEAK}}{\Sigma_i, \Gamma_i \vdash i; p' : \Sigma^*, \Gamma^* \mid C_i' \cup C_p \cup \{\Gamma_i' \leq \Gamma'\}}\ \textbf{SEQ}$$

Figure 11: Shorter typing derivation for $i; p'$ in the **WEAK** case.

also validates the predicate. Else, $[\![e]\!]_{\rho_j} = \bot$, in which case, b must be set. The semantics indicates that $[\![\mathsf{ms}]\!]_{\rho_j'} = -1$, which validates the predicate as well.

Either ms is changed identically in both executions or left as it was. This easily leads to $\rho_1' =_{\theta\Gamma_s'} \rho_2'$, and $\rho_1' =_{\theta\Gamma_n'} \rho_2'$ when it is required. Finally, $s_1' =^{\mathsf{ms}}_{\theta\Gamma'} s_2'$.

PROTECT $i = y = \mathsf{protect}(x, \mathsf{ms})$. The only type which differs is $\Gamma(y)$, and it will be the only one to be examined. If $\Gamma_n(x) = \mathsf{H}$, the result is trivial. Else, if b is set, then because of $\rho_1, b \vDash \mathsf{ms}$, we have $[\![\mathsf{ms}]\!]_{\rho_1} = -1$, which implies that $\rho_1'(y) = -1 = \rho_2'(y)$ and further ensures $\rho_1 =_{\theta\Gamma_s'} \rho_2$. Else b is unset, hence $\rho_1(\mathsf{ms}) \neq -1$, and $\rho_1'(y) = \rho_1(x)$. But because $\Gamma_n(x) = \mathsf{L}$, it means that $\rho_1(x) = \rho_2(x)$ and thus $\rho_1'(y) = \rho_2'(y)$. This leads to $\rho_1' =_{\theta\Gamma_n'} \rho_2'$. The equality $\rho_1' =_{\theta\Gamma_s'} \rho_2'$ also follows. The case of MSF is treated as in the **ASSIGN** rule. This finishes the case.

COND $i = \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2$. Let i also denote the chosen branch, and c_i be the chosen command. Let also $b_i = \mathsf{if}\ i = 1\ \mathsf{then}\ b\ \mathsf{else}\ \neg b$. This case requires a different proof. We shall prove that $\Gamma_0 = \Gamma, C_0 = C_i \cup C_p \cup \{\Gamma_i \leq \Gamma'\}$ and $\Sigma_0 = \Sigma_{|b_i}$ are satisfactory choices.

First, $\theta(\sigma_s) = \mathsf{L}$ and thus $[\![b]\!]_{\rho_1} = [\![b]\!]_{\rho_2}$, which implies the equality of leakage (after having discussed whether the directive is force or not). We have $p_j' = c_i; p'$, for $j = 1, 2$. The typing predicate is found as follows. The constraints ensure $\Gamma_i \leq \Gamma'$. Besides, a small case discussion leads to the following fact: for $j = 1, 2$, $\Sigma_1 \cap \Sigma_2 \subseteq \Sigma_j$. This allows to use a **WEAK** rule in order to type $c_i; p'$ directly, which is not shown here.

As for state equivalence, the predicates on memory are trivially verified, since the memory is left unchanged. MSF predicates are also easy: if $\Sigma \neq \mathsf{ms}$, then $\Sigma_0 = \bot$ and the result is trivial. Else, $\Sigma = \mathsf{ms}$ for some register ms, and $\Sigma_0 = \mathsf{ms}_{|b_i}$. Then, let's do the case $d = \mathsf{force}$, and the other case is similar. This implies $b_j' = \top$. Regardless the value of i, it is easy to notice that in that case $[\![b_i]\!]_{\rho_j'} = \top$. Then, $\rho_j', \top \vDash \mathsf{ms}_{|b_i}$ is verified. This ensures $s_1' =^{\Sigma_0}_{\theta\Gamma_0} s_2'$, and finishes the case.

WHILE $i = \mathsf{while}\ t\ \mathsf{do}\ c_0$. By Lemma 2, because $s_1 =_{\theta\Gamma} s_2$ and $\Gamma \leq \Gamma'$, we have $s_1 =^{\Sigma}_{\theta\Gamma'} s_2$. Consequently, by lemma 3, $[\![t]\!]_{\rho_1} = [\![t]\!]_{\rho_2}$. Equality of leakage follows from this. Besides, this also provides $s_1' =^{\Sigma'}_{\theta\Gamma'} s_2'$ since, as in **COND**, there is no change in the memory, and also, $\Sigma' \subseteq \Sigma$. MSF predicates are verified just as in the previous case.

There are two cases two consider, based on whether execution leaves the loop or not (depends on the value of e, as described in the semantics):

- the execution is leaving the loop ($e = \bot$): $p_i' = p'$. This, together with the fact that $s_1' =^{\Sigma_{|!t}}_{\theta\Gamma'} s_2'$ yields the result (as in the first cases), that is, $\Gamma_0 = \Gamma', C_0 = C_p, \Sigma_0 = \Sigma_{|!t}$ is satisfactory.
- the body of the loop is executed ($e = \top$): The idea is now to type $c_0; \mathsf{while}\ t\ \mathsf{do}\ c_0; p'$. This requires to choose other values, namely: $\Gamma_0 = \Gamma, \Sigma_0 = \mathsf{ms}_{|t}$ and $C_0 = C = C \cup C' \cup \{\Gamma' \leq \Gamma'\}$. The typing derivation itself is not shown here; please note that typing it requires building a new typing derivation for the while-loop itself.

□

Theorem 3 (Multi-step soundness)**.**

$$\begin{aligned} \overline{\forall} \quad & p, s_1 \xrightarrow[D]{O_1} p_1, q_1 \wedge p, s_2 \xrightarrow[D]{O_2} p_2, q_2 \wedge \\ & \Sigma, \Gamma \vdash p : \Sigma^*, \Gamma^* \mid C \wedge \theta \vDash C \wedge s_1 =^{\Sigma}_{\theta\Gamma} s_2 \\ & \qquad \Longrightarrow O_1 = O_2 \wedge p_1 = p_2 \\ & \wedge (\exists \Sigma_0, \Gamma_0, C' \quad \Sigma_0, \Gamma_0 \vdash p_1 : \Sigma^*, \Gamma^* \mid C' \\ & \qquad \wedge s_1' =^{\Sigma_0}_{\theta\Gamma_0} s_2' \wedge \theta \vDash C') \end{aligned}$$

Proof. The proof is by induction on the size $|D|$ of the list of directives (which is also the number of step of executions). If $|D| = 0$, only the **0-STEP** rule is possible, which yields the result trivially, with $\Gamma_0 = \Gamma, \Sigma_0 = \Sigma, C_0 = C$.

Else, we write:

$$p, s_1 \xrightarrow[d]{o_1} p_1', s_1' \xrightarrow[D']{O'} p_1, q_1$$
$$p, s_2 \xrightarrow[d]{o_2} p_2', s_2' \xrightarrow[D']{O'} p_2, q_2$$

and we apply theorem 2 to the two one-step executions thus exhibited. The corresponding Γ_0, Σ_0, C_0 can now serve in the premises of the induction hypothesis on the two remaining executions. This yields the result and finishes the proof. □