Relative Security: Formally Modeling and (Dis)Proving Resilience Against Semantic Optimization Vulnerabilities

Brijesh Dongol*, Matt Griffin*, Andrei Popescu† and Jamie Wright†
*Department of Computer Science, University of Surrey, UK Email: {b.dongol, matt.griffin}@surrey.ac.uk
†Department of Computer Science, University of Sheffield, UK Email: {a.popescu, jwright8}@sheffield.ac.uk

Abstract—**Meltdown and Spectre are vulnerabilities known as transient execution vulnerabilities, where an attacker exploits speculative execution (a semantic optimization present in most modern processors) to break confidentiality. We introduce *relative security*, a general notion of information-flow security that models this type of vulnerability by contrasting the leaks that are possible in a "vanilla" semantics with those possible in a different semantics, often obtained from the vanilla semantics via some optimizations. We describe incremental proof methods, in the style of Goguen and Meseguer's unwinding, both for proving and for disproving relative security, and deploy these to formally establish the relative (in)security of some standard Spectre examples. Both the abstract results and the case studies have been mechanized in the Isabelle/HOL theorem prover.**

1. Introduction

Meltdown [19] and Spectre [18] are transient execution vulnerabilities, which exploit timing-based side-channels caused by speculative execution optimizations, as present in almost all modern processors. Mitigating against such vulnerabilities is of significant importance for computer security. This paper proposes a new model-theoretic and proof-theoretic framework for describing and verifying resilience against transient execution attacks and beyond. We introduce *relative security* (§2), a general notion that focuses not on the absolute (information-flow) security of a system, but on the difference in security between a basic, "vanilla" system and a system that is enhanced to allow optimizations in the system's execution. More precisely, relative security expresses that there is *no* difference in information flow: Any leak occurring in the vanilla system can also occur in the optimization-enhanced one. We instantiate this notion to a programming language with speculative semantics (§3), yielding intuitive results on some standard example programs. While building on a rich literature that addresses this problem from a formal modeling perspective (§4), relative security's innovation is in natively capturing *fully interactive attackers* and *dynamic creation of secrets*, as required, e.g., by operating system processes. There are two key features enabling this: **1)** a fine-grained attacker model that allows secrets, attacker actions and attacker observations anywhere on the execution trace, and **2)** a view of *leaks as first-class citizens* that takes advantage of this fine granularity.

Our second contribution is general methods for *proving* and *disproving* relative security in an incremental fashion (§5), generalizing the unwinding method by Goguen and Meseguer [11]. On the proof front (§5.1–5.4), our method allows the incremental construction, from any two optimization-enhanced execution traces that exhibit a leak, of two counterpart vanilla traces that exhibit the same leak. The four traces are constrained by both "secrecy contracts" and "interaction contracts", which postulate local similarities and disimilarities between pairs of them.

Due to the complex dynamics between the four involved traces, the *disproof* front of relative security (§5.5) is also interesting and benefits from the idea of unwinding: A counterexample requires indicating a concrete leak in the optimization-enhanced system, followed by a proof employing a form of *secret-directed unwinding* that shows how this leak cannot be reproduced in the vanilla system. Both the proof and disproof methods have been validated on our example programs (§6).

The relative security framework, as well as the language semantics and examples, have been mechanized in the Isabelle/HOL prover [25] (§7). More details on our results and proofs can be found in an extended technical report [8].

2. Defining Relative Security

This section motivates and introduces our notion of relative security. We start with example programs exhibiting various features we would like to cover (§2.1), after which we develop the abstract framework: On top of system models capturing the program semantics (§2.2), we introduce *leakage models* which allow us to express the essence of relative security (§2.3). Then we instantiate these to more concrete *(state-wise) attacker models* (§2.4, §2.5), where leaks are defined from of secrets and attacker actions and observations.

2.1. Motivating examples

Next we discuss example C programs, including some standard ones from the Spectre literature [5, 17]. In all examples, we assume that the attacker controls the inputs and sees the outputs (unless specified otherwise), and can also infer the locations accessed for reading (as in Spectre attacks). The program in Listing 1 is vulnerable to a Spectre bounds-check-bypass attack. If the value of `x` is greater than or equal to `N`, under normal execution it is impossible for an attacker to infer the *value* of `a[x]` (which is needed to load a value from `b` into `t`). However, under the speculative execution

```c
unsigned fun1(unsigned x) {
  unsigned t = 0;
  if (x < N) {
    t = b[a[x] * 512]; }
  return t; }
```

Listing 1: Spectre Bounds Check Bypass (BCB)

```c
unsigned fun2(unsigned x) {
  unsigned t = 0;
  if (x < N) {
    _mm_lfence();
    t = b[a[x] * 512]; }
  return t; }
```

Listing 2: Fix 1 for Spectre BCB

```c
unsigned fun3(unsigned x) {
  unsigned t = 0;
  if (x < N) {
    unsigned v = a[x];
    _mm_lfence();
    t = b[v * 512]; }
  return t; }
```

Listing 3: Fix 2 for Spectre BCB

```c
unsigned fun4(unsigned x) {
  unsigned t = 0;
  if (x < N) {
    unsigned v = a[0];
    t = b[v * 512]; }
  return t; }
```

Listing 4: Conditionally secure BCB

```c
void fun5() {
  unsigned t = 0;
  unsigned x = 1;
  while (x != 0) {
    scanf("%u", &x);
    if (x < N) {
      unsigned v = a[x];
      _mm_lfence();
      t = b[v * 512];
      printf("%u", t); } } }
```

Listing 5: Secure interactive program

optimization, a branch misprediction enables this load to occur, potentially allowing the attacker to later perform a side-channel attack on the cache to infer the value of `a[x]`.

A suboptimal fix to this particular function is straightforward. As shown in Listing 2, one can introduce a load fence (`_mm_lfence`) instruction prior to loading from `b` and thus calculating the value of `a[x]`. This effectively resolves speculation by disallowing the processor from continuing execution until it can be certain that the branch will be taken, i.e., that `x` is indeed less than `N`.

A less obvious fix (given in Listing 3) is also possible. This program seemingly suffers from a transient execution vulnerability because speculative execution loads the value of `a[x]` into the cache, whereas non-speculative execution does not. However, since both `x` and the base address of `a` are known (or inferrable) to the attacker, loading `a[x]` does not leak any additional information under misprediction.

Now consider the program in Listing 4, discussed by Cheang et al. [5]. The authors refer to it as being *conditionally secure* because its susceptibility to a transient execution attack depends on the value of `N`. For `N = 0`, the program is insecure because `b[a[0] * 512]` can be loaded into the cache (thus leaking `a[0]`) only under misprediction.

Now suppose `N = 2` (i.e., `N` is some value greater than `0`), and consider a trace in which the attacker calls `fun4(3)`, triggering a misprediction and loading the value of `a[0]` into `v`. Should such an execution be ruled insecure? Unlike the program in Listing 1, here the value leaked from `a` is always from index `0`. Thus, the leak through misprediction described above is also possible through a normal execution, e.g., where the attacker chooses `x = 1`. Thus, the program does *not* have a transient execution vulnerability.

These examples have certain features in common: First, there are *secrets* to be protected—some parts of the memory (inside or outside the bounds of array `a`). Second, there are *observations* that an attacker can make—via standard channels such as function return or side channels, e.g., the affected cache. Finally, there are *actions* that an (active) attacker can take to interact with, and influence the execution of the program—via inputs to the functions.

While all the examples so far are only *end-to-end interactive*, i.e., take an input at the beginning and return an output at the end, this does not need to be the case. For example, Listing 5 shows a (possibly nonterminating) fully interactive program, which keeps inputting an integer value (via the `scanf` function) and accessing the arrays `a` and `b` based on it until the inputted value is 0. It also outputs a (possibly infinite) stream of elements from the array `b` (via the `printf` function). So here, depending on the particular setting, one may wish to assume that the attacker/observer continuously interacts with the program via both observations (printed values) and actions (scanned inputs)—and the Listing 5 program is still Spectre-secure under these assumptions.

Thus, actions and observations can take place not only at the execution's beginning and end respectively, but *fully interactively*, and possibly *in(de)finitely*. In some cases, this could also be true about secrets: One may wish to protect not only the initial memory, but also inputs that are under the control of trusted parties. Listing 6 shows a contrived example illustrating this: Now the program reads from both an untrusted and a trusted source, and one may wish to ensure that the input from the trusted source is not leaked. The call to a procedure named `writeOnSecretFile` illustrates the potential processing of some trusted input stored in `y` (here, together with that of untrusted input stored in `x`), which could influence a file on the disk; in this example, such an influence is harmless, since that file is assumed unobservable by the attacker.

In conclusion, we want to model fully interactive (both acting and observing) attackers and fully interactive secret

```
void fun6() {
  unsigned t = 0;
  unsigned x = 1;
  while (x != 0) {
    x = getUntrustedInput();
    unsigned y = getTrustedInput();
    if (x < N) {
      unsigned v = a[x];
      writeOnSecretFile(x,y);
      _mm_lfence();
      t = b[v * 512];
      printf("%u", t); } } }
```

Listing 6: Secure secret-interactive program

uploading, while also allowing interaction to take place in(de)finitely, i.e., factoring in infinite executions.

2.2. System models

For a set A, a *sequence* over A is an item in $\mathsf{Seq}(A) = A^* \cup A^{\mathbb{N}}$, i.e., a finite or infinite list of elements from A.

A *system model* is a triple $\mathcal{SM} = (\mathsf{State}, \mathsf{istate}, \Rightarrow)$, consisting of: **1)** a set State of *states*, ranged over by s, **2)** a predicate $\mathsf{istate} : \mathsf{State} \to \mathsf{Bool}$ that describes the *initial states*, and **3)** a binary relation $\Rightarrow : \mathsf{State} \times \mathsf{State} \to \mathsf{Bool}$ on states called the *transition relation*.

The next concepts are relative to a system model $\mathcal{SM} = (\mathsf{State}, \mathsf{istate}, \Rightarrow)$. We say that a state $s \in \mathsf{State}$ *is final*, written $\mathsf{final}(s)$, when there is no transition out of s, i.e, $\neg \exists s'. \, s \;\Rightarrow\; s'$. An *(execution) trace* is a nonempty (finite or infinite) maximal sequence of states $s_0 \, s_1 \ldots \in \mathsf{Seq}(\mathsf{State})$ such that $\mathsf{istate}(s_0)$ holds and $s_i \Rightarrow s_{i+1}$ for all i. Maximality refers to the suffix relation; it is equivalent to saying that, if the sequence is finite then its last state is final. We let $\mathsf{Trace} \subseteq \mathsf{Seq}(\mathsf{State})$, ranged over by π, ρ, be the set of traces, $\mathsf{Trace}^{\mathsf{fin}}$ be the subset of Trace consisting of the finite traces only.

2.3. Very abstract relative security: leakage models

We fix two system models $\mathcal{SM}_{\mathsf{van}} = (\mathsf{State}_{\mathsf{van}}, \mathsf{istate}_{\mathsf{van}}, \Rightarrow_{\mathsf{van}})$ and $\mathcal{SM}_{\mathsf{opt}} = (\mathsf{State}_{\mathsf{opt}}, \mathsf{istate}_{\mathsf{opt}}, \Rightarrow_{\mathsf{opt}})$ such as $\mathsf{State}_{\mathsf{van}} \subseteq \mathsf{State}_{\mathsf{opt}}$. We will call $\mathcal{SM}_{\mathsf{van}}$ the *vanilla* system model, and refer to its states, transition relation etc. as vanilla states, vanilla transition etc. Its set of traces is denoted by $\mathsf{Trace}_{\mathsf{van}}$. The vanilla system model represents the plain system, featuring no optimization—which will act as reference for our (relative) notion of security. Moreover, we will call $\mathcal{SM}_{\mathsf{opt}}$ the *optimization-enhanced* system model, since it will stand for the system that has been optimized in various ways, e.g., with speculative or out-of-order executions. Its set of traces is denoted by $\mathsf{Trace}_{\mathsf{opt}}$. We will use the following short names: "vtrace" for "vanilla trace" (i.e., an element of $\mathsf{Trace}_{\mathsf{van}}$), and "otrace" for "optimization-enhanced trace" (i.e., an element of $\mathsf{Trace}_{\mathsf{opt}}$).

We are interested in expressing the information-flow security of $\mathcal{SM}_{\mathsf{opt}}$ relative to that of $\mathcal{SM}_{\mathsf{van}}$, in order to assess whether the optimizations have introduced further vulnerabilities (as is the case with Spectre). However, we do not just want to check the implication "if $\mathcal{SM}_{\mathsf{van}}$ does not leak then $\mathcal{SM}_{\mathsf{opt}}$ does not leak either", because that would be too coarse: We wish to allow $\mathcal{SM}_{\mathsf{van}}$ to have some (presumably acceptable, or at least known) leaks, and to check that $\mathcal{SM}_{\mathsf{opt}}$ has no *additional* leaks.

What we seem to need for this is the ability to talk *leak-wise*, i.e., to express not just the *absence* of any leak (as done by traditional notions such as noninterference), but explicitly the *very notion* of a leak. To this end, we introduce leakage models: Given any system model $\mathcal{SM} = (\mathsf{State}, \mathsf{istate}, \Rightarrow)$, a *leakage model for* $\mathcal{SM}$ is a pair $(\mathsf{Leak}, \mathsf{leakVia})$, where Leak, ranged over by l, is a set of entities called *leaks*, and leakVia is a predicate in $\mathsf{Trace} \times \mathsf{Trace} \times \mathsf{Leak} \to \mathsf{Bool}$. We think of $\mathsf{leakVia}(\pi_1, \pi_2, l)$ as expressing that the traces π_1 and π_2 exhibit the leak l. Indeed, it is well known [6] that whatever the leaks are, one requires two traces (i.e., two alternative executions) rather than just one to exhibit a leak.

Note that we work very abstractly, gradually instantiating our concepts. For now, we leave the notion of a leak unspecified. In the next subsection, §2.4, we will get more concrete, taking leaks to be pairs of sequences of secrets produced by alternative execution traces—which, for suitable choices of the notion of secret, recovers what is typically taken to constitute a leak in a language-based setting [28]. In §3, when we have at our disposal some concrete system models given by a programming language semantics, we further instantiate the secrets to be the initial memories and inputs and outputs over certain trusted channels.

We now have all the ingredients for an abstract, leak-centric definition of relative security:

Def. 1 Let $\mathcal{LM}_{\mathsf{van}} = (\mathsf{Leak}, \mathsf{leakVia}_{\mathsf{van}})$ and $\mathcal{LM}_{\mathsf{opt}} = (\mathsf{Leak}, \mathsf{leakVia}_{\mathsf{opt}})$ be leakage models for $\mathcal{SM}_{\mathsf{van}}$ and $\mathcal{SM}_{\mathsf{opt}}$ respectively (having the same set of leaks Leak). We say that $(\mathcal{SM}_{\mathsf{opt}}, \mathcal{LM}_{\mathsf{opt}})$ satisfies *relative security* w.r.t. $(\mathcal{SM}_{\mathsf{van}}, \mathcal{LM}_{\mathsf{van}})$, written $(\mathcal{SM}_{\mathsf{opt}}, \mathcal{LM}_{\mathsf{opt}}) \geq_{\checkmark} (\mathcal{SM}_{\mathsf{van}}, \mathcal{LM}_{\mathsf{van}})$, when the following holds:

$$\forall l \in \mathsf{Leak}_{\mathsf{opt}}.\ \forall \pi_1, \pi_2 \in \mathsf{Trace}_{\mathsf{opt}}.\ \mathsf{leakVia}_{\mathsf{opt}}(\pi_1, \pi_2, l) \longrightarrow \exists \hat{\pi}_1, \hat{\pi}_2 \in \mathsf{Trace}_{\mathsf{van}}.\ \mathsf{leakVia}_{\mathsf{van}}(\hat{\pi}_1, \hat{\pi}_2, l)$$

We say that $(\mathcal{SM}_{\mathsf{opt}}, \mathcal{LM}_{\mathsf{opt}})$ satisfies *finitary relative security* w.r.t. $(\mathcal{SM}_{\mathsf{van}}, \mathcal{LM}_{\mathsf{van}})$, written $(\mathcal{SM}_{\mathsf{opt}}, \mathcal{LM}_{\mathsf{opt}}) \geq^{\mathsf{fin}}_{\checkmark} (\mathcal{SM}_{\mathsf{van}}, \mathcal{LM}_{\mathsf{van}})$, when the above property holds when restricted to finite traces, i.e., replacing $\mathsf{Trace}_{\mathsf{opt}}$ with $\mathsf{Trace}^{\mathsf{fin}}_{\mathsf{opt}}$ and $\mathsf{Trace}_{\mathsf{van}}$ with $\mathsf{Trace}^{\mathsf{fin}}_{\mathsf{van}}$.

This definition expresses that, for the given notion of leak, the optimization-enhanced system $\mathcal{SM}_{\mathsf{opt}}$ does not exhibit any leaks besides those already exhibited by the vanilla system $\mathcal{SM}_{\mathsf{van}}$. (Thinking of $\checkmark$ as expressing security, the notation $\geq_{\checkmark}$ suggests an "at least as secure as" reading.) The notations $\hat{\pi}_1$ and $\hat{\pi}_2$ for vanilla traces remind of their dependency on the otraces π_1 and π_2—with the caveat that each of the two vtraces may depend on *both* π_1 and π_2.

We believe the finitary (i.e., termination-conditioned) variant of relative security $\geq^{\mathsf{fin}}_{\checkmark}$ is of interest for both historic reasons (since often the discussion in the literature is restricted to finite traces, e.g., [13]) and pragmatic reasons

(since, as we shall see, finitary security is amenable to a simpler unwinding proof method). $\geq_{\checkmark}^{\text{fin}}$ is the same as $\geq_{\checkmark}$ for terminating programs such as the ones in §2.1's Listings 1–4, but $\geq_{\checkmark}$ is more suitable for possibly nonterminating interactive programs as in our Listings 5 and 6.

2.4. Less abstract relative security: attacker models

Our current definition of relative security is parameterized by leaks. But still, what *is* a leak more concretely? To give a plausible answer, remember the key ingredients identified in our examples (§2.1): secret uploading and observer interaction, and our desire to capture fully interactive versions of these. These lead us to the next definition. Given a system model $(\mathsf{State}, \mathsf{istate}, \Rightarrow)$, an *attacker model* for it is a tuple $(\mathsf{Sec}, \mathsf{S}, \mathsf{Obs}, \mathsf{O}, \mathsf{Act}, \mathsf{A})$ where:

- Sec, Obs and Act are sets of items called *secrets*, *observations* and *actions* respectively;
- $\mathsf{S} : \mathsf{Trace} \to \mathsf{Seq}(\mathsf{Sec})$, $\mathsf{O} : \mathsf{Trace} \to \mathsf{Seq}(\mathsf{Obs})$ and $\mathsf{A} : \mathsf{Trace} \to \mathsf{Seq}(\mathsf{Act})$ are functions called the *secrecy*, *observation* and *action* functions, respectively.

Thus, attacker models indicate what needs protection (the secrets) and what attacker actions / observations are available.

The functions S, O and A provide a natural notion of leak. We take Leak to be $\mathsf{Seq}(\mathsf{Sec}) \times \mathsf{Seq}(\mathsf{Sec})$ and $\mathsf{leakVia}(\pi_1, \pi_2, (\sigma_1, \sigma_2))$ to be $\mathsf{S}(\pi_1) = \sigma_1 \land \mathsf{S}(\pi_2) = \sigma_2 \land \mathsf{A}(\pi_1) = \mathsf{A}(\pi_2) \land \mathsf{O}(\pi_1) \neq \mathsf{O}(\pi_2)$, which means:

- π_1 and π_2 have the sequences of secrets σ_1 and σ_2;
- the attacker took the *same* actions during π_1 and π_2 ...
- which led to the attacker making *different* observations.

$\mathsf{leakVia}(\pi_1, \pi_2, (\sigma_1, \sigma_2))$ therefore says that, via π_1 and π_2, the attacker can observationally distinguish between σ_1 and σ_2. Note that it was crucial to require that the distinction be made while the attacker takes the *same actions*—otherwise it would not tell us anything about the secrets (as it could simply be a consequence of the different actions).

Thus, any attacker model for $\mathcal{SM}$ induces a leakage model for $\mathcal{SM}$. It is worth spelling out what the definition of relative security becomes in this more concrete setting:

Def. 2 Let $\mathcal{AM}_{\mathsf{van}} = (\mathsf{Sec}, \mathsf{S}_{\mathsf{van}}, \mathsf{Obs}_{\mathsf{van}}, \mathsf{O}_{\mathsf{van}}, \mathsf{Act}_{\mathsf{van}}, \mathsf{A}_{\mathsf{van}})$ and $\mathcal{AM}_{\mathsf{opt}} = (\mathsf{Sec}, \mathsf{S}_{\mathsf{opt}}, \mathsf{Obs}_{\mathsf{opt}}, \mathsf{O}_{\mathsf{opt}}, \mathsf{Act}_{\mathsf{opt}}, \mathsf{A}_{\mathsf{opt}})$ be attacker models for $\mathcal{SM}_{\mathsf{van}}$ and $\mathcal{SM}_{\mathsf{opt}}$ respectively (with same set of secrets Sec). We say that $(\mathcal{SM}_{\mathsf{opt}}, \mathcal{AM}_{\mathsf{opt}})$ satisfies *relative security* w.r.t. $(\mathcal{SM}_{\mathsf{van}}, \mathcal{AM}_{\mathsf{van}})$, written $(\mathcal{SM}_{\mathsf{opt}}, \mathcal{AM}_{\mathsf{opt}}) \geq_{\checkmark} (\mathcal{SM}_{\mathsf{van}}, \mathcal{AM}_{\mathsf{van}})$, when:

$$
\begin{array}{l}
\forall \sigma_1, \sigma_2 \in \mathsf{Seq}(\mathsf{Sec}).\ \forall \pi_1, \pi_2 \in \mathsf{Trace}_{\mathsf{opt}}. \\
\quad \mathsf{S}_{\mathsf{opt}}(\pi_1) = \sigma_1 \ \land\ \mathsf{S}_{\mathsf{opt}}(\pi_2) = \sigma_2 \ \land \\
\quad \mathsf{A}_{\mathsf{opt}}(\pi_1) = \mathsf{A}_{\mathsf{opt}}(\pi_2) \ \land\ \mathsf{O}_{\mathsf{opt}}(\pi_1) \neq \mathsf{O}_{\mathsf{opt}}(\pi_2) \\
\quad \longrightarrow \\
\quad \exists \hat{\pi}_1, \hat{\pi}_2 \in \mathsf{Trace}_{\mathsf{van}}.\ \mathsf{S}_{\mathsf{van}}(\hat{\pi}_1) = \sigma_1 \ \land\ \mathsf{S}_{\mathsf{van}}(\hat{\pi}_2) = \sigma_2 \ \land \\
\qquad \mathsf{A}_{\mathsf{van}}(\hat{\pi}_1) = \mathsf{A}_{\mathsf{van}}(\hat{\pi}_2) \ \land\ \mathsf{O}_{\mathsf{van}}(\hat{\pi}_1) \neq \mathsf{O}_{\mathsf{van}}(\hat{\pi}_2)
\end{array}
$$

Finitary relative security, $(\mathcal{SM}_{\mathsf{opt}}, \mathcal{AM}_{\mathsf{opt}}) \geq_{\checkmark}^{\text{fin}} (\mathcal{AM}_{\mathsf{van}}, \mathcal{LM}_{\mathsf{van}})$, is again defined by restricting to finite traces.

The above just expands the aforementioned construction of leakage models from attacker models, so Def. 2 is a particular case of Def. 1. Its conclusion can be reformulated without explicitly quantifying over secrets as follows:

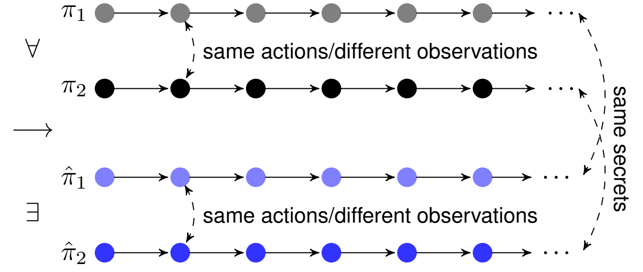

Fig. 1: Visualizing relative security

$$
\begin{array}{ll}
\forall \pi_1, \pi_2 \in \mathsf{Trace}_{\mathsf{opt}}. & \\
\mathsf{A}_{\mathsf{opt}}(\pi_1) = \mathsf{A}_{\mathsf{opt}}(\pi_2) \ \land\ \mathsf{O}_{\mathsf{opt}}(\pi_1) \neq \mathsf{O}_{\mathsf{opt}}(\pi_2) & \\
\longrightarrow & (\bigstar) \\
\exists \hat{\pi}_1, \hat{\pi}_2 \in \mathsf{Trace}_{\mathsf{van}}. & \\
\quad \mathsf{S}_{\mathsf{van}}(\hat{\pi}_1) = \mathsf{S}_{\mathsf{van}}(\pi_1) \ \land\ \mathsf{S}_{\mathsf{van}}(\hat{\pi}_2) = \mathsf{S}_{\mathsf{van}}(\pi_2) \ \land & \\
\quad \mathsf{A}_{\mathsf{van}}(\hat{\pi}_1) = \mathsf{A}_{\mathsf{van}}(\hat{\pi}_2) \ \land\ \mathsf{O}_{\mathsf{van}}(\hat{\pi}_1) \neq \mathsf{O}_{\mathsf{van}}(\hat{\pi}_2) &
\end{array}
$$

The $(\bigstar)$ formulation, which is the one we will prefer, facilitates an intuitive reading of relative security, as visualized in Fig. 1: Provided the otraces π_1 and π_2 have the same actions and different observations, the vtraces $\hat{\pi}_1$, $\hat{\pi}_2$ must be proved to exist such that they have the same secrets as π_1 and π_2 respectively, and also have between each other the same actions and different observations.

In line with our aforementioned gradual instantiation approach, for now we have left S, A and O unspecified. In the next section, §2.5, we make these more concrete by assuming they operate "state-wise", i.e., the secrets, actions and observations are produced locally from each state of a traces. (Another natural choice would be to assume they operate transition-wise rather than state-wise; these two choices are equivalent, since states can also encode transition information.) For our §2.1 motivating examples, when applied to a trace: S will give the initial memory, as well as any potential secrets received or sent during the execution (e.g., on the trusted input channel in Listing 6); moreover, A will give any inputs on untrusted channels; finally, O will give any returned or printed values via untrusted channels, and the memory locations accessed for reading—see §3.3.

2.5. State-wise attacker models

Next, we will consider an assumption on attacker models that makes them even less abstract (thus bringing us closer to considering concrete examples). Given a system model $\mathcal{SM} = (\mathsf{State}, \mathsf{istate}, \Rightarrow)$, an attacker model $\mathcal{AM} = (\mathsf{Sec}, \mathsf{S}, \mathsf{Obs}, \mathsf{O}, \mathsf{Act}, \mathsf{A})$ for it is said to be *state-wise* when there exist the predicates and functions $\mathsf{isSec} : \mathsf{State} \to \mathsf{Bool}$, $\mathsf{getSec} : \mathsf{State} \to \mathsf{Sec}$, $\mathsf{isInt} : \mathsf{State} \to \mathsf{Bool}$, $\mathsf{getObs} : \mathsf{State} \to \mathsf{Obs}$, and $\mathsf{getAct} : \mathsf{State} \to \mathsf{Act}$ that define the functions S, O and A state-wise as follows. For any trace $\pi = s_0\, s_1 \ldots$:

- Let $s_{i_0} s_{i_1} \dots$ for $i_0 < i_1 < \dots$ be its subsequence consisting of states where isSec holds. We define $\mathsf{S}(\pi)$ as $\mathsf{getSec}(s_{i_0})\,\mathsf{getSec}(s_{i_1}) \dots$
- Let $s_{j_0} s_{j_1} \dots$ for $j_0 < j_1 < \dots$ be its subsequence consisting of the states where isInt holds. We define $\mathsf{O}(\pi)$ as $\mathsf{getObs}(s_{j_0})\,\mathsf{getObs}(s_{j_1}) \dots$ and $\mathsf{A}(\pi)$ as $\mathsf{getAct}(s_{j_0})\,\mathsf{getAct}(s_{j_1}) \dots$

What is being required above is that the trace functions S, O and A are defined by "filtermap", filtering with a predicate and mapping with a getter function.

We think of the predicates isSec and isInt as determining whether (the next transition from) a state uploads a secret, and is a point of interaction (observation or action), respectively. For example, in our concrete programming language models, the program counter stored in the state will determine the next statement to be executed, and therefore isSec and isInt will check whether this next statement is secret-uploading or interaction-producing. (We do not require isSec and isInt to be disjoint, although for some systems this can be a reasonable assumption for *a priori* excluding obvious leaks.) Moreover, we think of the functions getSec, getObs and getAct as actually extracting that particular secret, observation or action.

3. Language-Based Instantiation

Next we describe a concrete instance of relative security (§3.3), via a programming language with speculative execution (§3.1) that can express our running examples (§3.2).

3.1. The IMP Language with Speculative Semantics

We introduce IMP, a simple language that exhibits interesting security aspects stemming from speculative execution. Its speculative semantics follows the ideas of Cheang et al. [5], maintaining runtime configurations for nested speculative executions—and only for those that result from *misprediction*, since they are the only security-relevant ones.

Syntax. The set Lit of *literals*, ranged over by i, j etc., is taken to be $\mathbb{N}$. Var, the set of *(scalar-)variables*, ranged over by x, y, z etc., is a fixed countably infinite set; and so is AVar, the set of *array-variables*, ranged over by a. Op is the set of binary arithmetic operators, e.g., $+, *$, etc; COp, that of binary comparison operators, e.g., $<, ==$, etc; and BOp that of binary boolean operators, e.g., $\wedge, \vee$, etc. ICh, ranged over by ich, and OCh, ranged over by och, are sets of input and output channels, respectively.

The sets of (arithmetic) expressions, boolean expressions and commands are defined by the grammar:

$$\begin{array}{rcl}
\mathsf{Exp} & ::= & \mathsf{Lit} \mid \mathsf{Var} \mid \mathsf{AVar}[\mathsf{Exp}] \mid \mathsf{Exp}\ \mathsf{Op}\ \mathsf{Exp} \\
\mathsf{BExp} & ::= & \mathsf{true} \mid \mathsf{false} \mid \mathsf{Exp}\ \mathsf{COp}\ \mathsf{Exp} \mid \\
 & & \mathsf{not}\ \mathsf{BExp} \mid \mathsf{BExp}\ \mathsf{BOp}\ \mathsf{BExp} \\
\mathsf{Com} & ::= & \mathsf{Start} \mid \mathsf{Input}_{\mathsf{ICh}}\ \mathsf{Var} \mid \mathsf{Output}_{\mathsf{OCh}}\ \mathsf{Exp} \mid \\
 & & \mathsf{Fence} \mid \mathsf{Var} = \mathsf{Exp} \mid \mathsf{AVar}[\mathsf{Exp}] = \mathsf{Exp} \mid \\
 & & \mathsf{Jump}\ \mathbb{N} \mid \mathsf{IfJump}\ \mathsf{BExp}\ \mathbb{N}\ \mathbb{N}
\end{array}$$

We let e range over Exp, b over BExp, and c over Com.

StartOrFenceOrOutput
$$\frac{c_{pc} \in \{\mathsf{Start}, \mathsf{Fence}\} \cup \{\mathsf{Output}_{och}\ e \mid e \in \mathsf{Exp}\}}{((pc, \mu), inp) \Rightarrow_B ((pc+1, \mu), inp)}$$

VarAssign
$$\frac{c_{pc} = (x = e)}{((pc, \mu), inp) \Rightarrow_B ((pc+1, \mu[x \leftarrow [\![e]\!](\mu)]), inp)}$$

AVarAssign
$$\frac{c_{pc} = (a[e] = e')}{((pc, \mu), inp) \Rightarrow_B ((pc+1, \mu[(a, [\![e]\!](\mu)) \leftarrow [\![e']\!](\mu)]), inp)}$$

Input
$$\frac{c_{pc} = (\mathsf{Input}_{ich}\ x) \qquad inp_{ich} = i \cdot is'}{((pc, \mu), inp) \Rightarrow_B ((pc+1, \mu[x \leftarrow i]), inp[ich \leftarrow is'])}$$

Jump
$$\frac{c_{pc} = (\mathsf{Jump}\ pc')}{((pc, \mu), inp) \Rightarrow_B ((pc', \mu), inp)}$$

IfJump
$$\frac{c_{pc} = (\mathsf{IfJump}\ b\ pc_1\ pc_2) \qquad pc' = (\text{if } [\![b]\!](\mu) \text{ then } pc_1 \text{ else } pc_2)}{((pc, \mu), inp) \Rightarrow_B ((pc', \mu), inp)}$$

Fig. 2: Basic semantics for program $P = c_0; \dots; c_n$. We implicitly assume $pc \leq n$ as a condition in each rule.

Thus, IMP has the standard basic mechanisms for manipulating scalar and array variables, and (un)conditional jumps, Jump and IfJump, as control structures. It is also an I/O interactive language, accepting inputs on input channels and producing outputs on output channels. A *program* $P = c_0; c_1; \dots; c_n$ is a nonempty list of commands where $c_0 = \mathsf{Start}$. Prog denotes the set of programs.

The set Val of *values*, ranged over by v, w etc., is $\mathbb{N}$. Loc, the set of *locations*, ranged over by l, is also $\mathbb{N}$.

Basic semantics. Fig. 2 shows the *basic (small-step) semantics*, denoted $\Rightarrow_B$, parameterized by a fixed program $P = c_0; \dots; c_n$. It maintains input streams and memories, which are consumed and respectively updated while the program counter moves through the program's list of commands.

In detail, $\Rightarrow_B$ is a relation between pairs (cfg, inp) where $inp : \mathsf{ICh} \to \mathsf{Seq}(\mathsf{Val})$ is an input-channel indexed family of *input streams*, and cfg is a *configuration*, i.e., a pair (pc, μ):

- $pc \in \mathbb{N}$ is the *program counter (PC)*, pointing to the pc'th statement in the program, c_{pc}.
- μ is a *memory*, i.e., a triple (vs, avs, hp) where $vs : \mathsf{Var} \to \mathsf{Val}$ is a *variable store* assigning values to the (scalar) variables, $avs : \mathsf{AVar} \to \mathsf{Loc} \times \mathbb{N}$ is a *array-variable store* assigning to each array-variable its starting location in the heap and its size, and $hp : \mathsf{Loc} \to \mathsf{Val}$ is a *heap* assigning values to locations.

If $cfg = (pc, (vs, avs, hp))$, we let $\mathsf{pcOf}(cfg) = pc$. We use standard notation for updates, e.g., $\mu[x \leftarrow v]$, and expression evaluation. $[\![e]\!](\mu)$. $A_\perp$ denotes the extension of

$$\frac{(cfg, inp) \Rightarrow_B (cfg', inp')}{(cfg, inp, L) \Rightarrow_N (cfg', inp', L \cup \mathsf{readLocs}(cfg))} \text{ STANDARD}$$

Fig. 3: Normal semantics for program $P = c_0; \ldots; c_n$. We implicitly assume $pc \leq n$.

$$\frac{c_{pc} = (\mathsf{IfJump}\ b\ pc_1\ pc_2) \qquad pc' = (\text{if } [\![b]\!](\mu) \text{ then } pc_2 \text{ else } pc_1)}{((pc, \mu), inp) \Rightarrow_M ((pc', \mu), inp)} \text{ IFJUMPMISPRED}$$

$$\frac{\begin{array}{c}\neg\, \mathsf{isCond}(cfg_k) \vee \neg\, \mathsf{mispred}(ps, pcs) \\ (k > 0 \longrightarrow \neg\, \mathsf{isIOorFence}(cfg_k) \wedge \neg\, \mathsf{resolve}(ps, pcs)) \\ (cfg_k, inp) \Rightarrow_B (cfg', inp') \\ C' = cfg_0 \cdot \ldots \cdot cfg_{k-1} \cdot cfg' \qquad L' = L \cup \mathsf{readLocs}(cfg_k)\end{array}}{(ps, cfg_0 \cdot \ldots \cdot cfg_k, inp, L) \Rightarrow_S (ps, C', inp', L')} \text{ STANDARD}$$

$$\frac{\begin{array}{cc}\mathsf{isCond}(cfg_k) & \mathsf{mispred}(ps, pcs) \\ (cfg_k, inp) \Rightarrow_B (cfg', inp') & (cfg_k, inp) \Rightarrow_M (cfg'', inp'') \\ C' = cfg_0 \cdot \ldots \cdot cfg_{k-1} \cdot cfg' \cdot cfg'' & L' = L \cup \mathsf{readLocs}(cfg_k)\end{array}}{(ps, cfg_0 \cdot \ldots \cdot cfg_k, inp, L) \Rightarrow_S (\mathsf{update}(ps, pcs), C', inp', L')} \text{ MISPRED}$$

$$\frac{k > 0 \qquad \mathsf{resolve}(ps, pcs) \qquad C' = cfg_0 \cdot \ldots \cdot cfg_{k-1}}{(ps, cfg_0 \cdot \ldots \cdot cfg_k, inp, L) \Rightarrow_S (\mathsf{update}(ps, pcs), C', inp, L)} \text{ RESOLVE}$$

$$\frac{k > 0 \qquad \neg\, \mathsf{resolve}(ps, pcs) \qquad \mathsf{isFence}(cfg_k)}{(ps, cfg_0 \cdot \ldots \cdot cfg_k, inp, L) \Rightarrow_S (pcs, cfg_0, inp, L)} \text{ FENCE}$$

Fig. 4: Speculative semantics for program $P = c_0; \ldots; c_n$ under misprediction oracle ($\mathsf{PState}, \mathsf{mispred}, \mathsf{resolve}, \mathsf{update}$). Each time, we implicitly assume $pc \leq n$ and $pcs = pc_0 \cdot \ldots \cdot pc_k$ where $pc_i = \mathsf{pcOf}(cfg_i)$ for $i \in \{0, \ldots, k\}$.

a set A with an "undefined" element $\bot$. The *output of a configuration*, $\mathsf{outOf}(pc, \mu) \in (\mathsf{OCh} \times \mathsf{Val})_\bot$, is $(och, [\![e]\!](\mu))$ if c_{pc} has the form $\mathsf{Output}_{och}\ e$, and $\bot$ otherwise.

The read location. For modeling Spectre vulnerabilities, we will record memory reads (as in [5]). We let $\mathsf{readLocs}(pc, \mu)$ be the (possibly empty) set of locations that are read by the current command c_{pc}—computed from all sub-expressions of c_{pc} of the form $a[e]$. For example, if c_{pc} is the assignment $x = a[b[3]]$, then readLocs returns two locations: counting from 0, the 3rd location of b and the $b[3]$'th location of a.

Normal semantics. Fig. 3 shows the "normal" semantics $\Rightarrow_N$, which is the basic semantics extended to accumulate the read locations, hence account for cache side-channels.

Speculative semantics. Finally, Fig. 4 shows the *speculative semantics* $\Rightarrow_S$, which augments normal semantics with speculative steps that go wrong, along a *misprediction* (taking the wrong branch). In addition to the program $P = c_0; \ldots; c_n$, $\Rightarrow_S$ is parameterized by a *misprediction oracle*, i.e., a tuple ($\mathsf{PState}, \mathsf{mispred}, \mathsf{resolve}, \mathsf{update}$) where: PState is a set of *predictor states* ranged over by ps, $\mathsf{mispred} : \mathsf{PState} \times \{0, \ldots, n\}^* \to \mathsf{Bool}$, $\mathsf{resolve} : \mathsf{PState} \times \{0, \ldots, n\}^* \to \mathsf{Bool}$, and $\mathsf{update} : \mathsf{PState} \times \{0, \ldots, n\}^* \to \mathsf{PState}$.

The oracle decides for when misprediction occurs thus triggering a new speculation (the mispred predicate), and for when a speculation is being resolved (the resolve predicate). Both occurrence and resolution depend on the predictor state (which evolves via the function update) and on the list of PCs of the nested speculations, an element of $\{0, \ldots, n\}^*$. Indeed, the semantics allows for nested speculation: The runtime environment can mispredict thus taking a wrong branch, and while running on that wrong branch it can mispredict again, and so on. Resolution, i.e., the act of the runtime environment figuring out that the prediction was wrong and reverting the corresponding speculative execution, occurs when the oracle dictates that, via resolve.

The speculative semantics $\Rightarrow_S$ operates on *(multi-speculative) states*, i.e., tuples $s = (ps, cfgs, inp, L)$ where $ps \in \mathsf{PState}$ is the current predictor state, $cfgs$ is a non-empty list of configurations. inp is a family of input streams and L is the set of locations read so far. We think of $cfgs$ as a stack of configurations, one for each *speculation level* in a nested speculative execution. At level 0 we have the configuration cfg_0 for normal, non-speculative execution.

At each moment, only the top of the configuration stack, cfg_k, is active. One type of transitions that cfg_k can take is standard transitions, shown in the rule STANDARD. These occur only when (1) the misprediction option is not on the table (either because the command of cfg_k is not a conditional, or because the oracle does not demand a misprediction at this time) and (2) if in speculation mode, i.e., $k > 0$, no other blocking factors occur.

If the command of cfg_k is a conditional and the oracle demands misprediction, then a new mispredicting speculation is launched—as shown in the rule MISPRED. In this case, cfg_k takes a normal execution step on the correct branch (yielding the configuration cfg') while at the same time cfg_k takes a step on the wrong branch (yielding the configuration cfg'' at speculation level $k + 1$, which becomes the active configuration). The step on the wrong branch is achieved using a dual of the $\Rightarrow_B$ transition for conditional commands, denoted $\Rightarrow_M$ (where M stands for "misprediction")—also shown in Fig. 4, as the rule IFJUMP-MISPRED. The fact that, in the semantics, the new speculation execution moves on the wrong branch at the same time with execution one level below moving on the correct branch is a succinct way of expressing that the new speculation involves misprediction (as opposed to correct prediction).

When in speculative mode ($k > 0$) and the oracle demands it, a resolution occurs: The k-level speculation is reverted, i.e., the configuration cfg_k is dropped and cfg_{k-1} re-becomes the active configuration —as shown by the rule RESOLVE. Finally, when in speculative mode and encountering a Fence command, the entire stack of nested speculations is reverted, leaving only the configuration at level 0 (for normal execution)—as shown by the rule FENCE.

3.2. Representing the running examples in IMP

We assume $\mathsf{ICh} = \mathsf{OCh} = \{\mathsf{T}, \mathsf{U}\}$, where T and U represent a trusted and an untrusted channel, respectively. Moreover, we assume a binary operation $\mathsf{F} \in \mathsf{Op}$ whose semantics is some (fixed but) unspecified function in $\mathsf{Val}^2 \to \mathsf{Val}$.

Our motivating examples from §2.1 are expressed in IMP as depicted in Fig. 5. Expectedly, the if statements are modeled by IfJump, and the while loops by IfJump in conjunction with unconditional Jump, In fun1–fun5 we use only the untrusted channel U for inputs and outputs—thus matching our §2.1's assumption that the attacker controls the inputs and sees the outputs. In fun6 we use both the trusted and untrusted channels, matching our assumption that the attacker controls only specific inputs and outputs. Since our language lacks procedures, we use $\mathsf{Output}_\mathsf{T}\ (\mathsf{F}(x, y))$ to encode the writeOnSecretFile(x,y) procedure of fun6.

3.3. Instantiating relative security to IMP

We fix an IMP program $P = c_0; \ldots; c_n$, and some initial memory μ_0, inputs inp_0 and predictor state ps_0. We first instantiate the system models $\mathcal{SM}_\mathsf{van}$ and $\mathcal{SM}_\mathsf{opt}$:

- $\mathsf{State}_\mathsf{van}$ consists of non-speculative states, i.e., triples (cfg, inp, L); $\Rightarrow_\mathsf{van}$ is the normal semantics, $\Rightarrow_N$; and $\mathsf{istate}_\mathsf{van}(cfg, inp, L)$ says that $cfg = (0, \mu_0)$, $inp = inp_0$ and $L = \emptyset$.
- $\mathsf{State}_\mathsf{opt}$ consist of the (multi-speculative) states; $\Rightarrow_\mathsf{opt}$ is $\Rightarrow_S$; and $\mathsf{istate}_\mathsf{opt}(s)$ for $s = (ps, cfgs, inp, L)$ says that $ps = ps_0$, $inp = inp_0$, $L = \emptyset$ and $cfgs$ is a (singleton) list consisting of one configuration, $(0, \mu_0)$.

Thus, both systems start with the specified initial memory and input stream, and no read locations or speculation.

For instantiating the state-wise attacker models $\mathcal{AM}_\mathsf{opt}$ and $\mathcal{AM}_\mathsf{van}$, we note that the non-speculative states in $\mathsf{State}_\mathsf{van}$ can be seen as particular cases of (multi-speculative) states in $\mathsf{State}_\mathsf{opt}$. Therefore, agreeing to take the operators for $\mathcal{AM}_\mathsf{van}$ as the restrictions of those for $\mathcal{AM}_\mathsf{opt}$, we will allow ourselves to only define the latter, while also omitting the subscript opt.

To determine isInt and getInt, we ask what the attacker's capabilities are. Of course, the attacker controls/observes untrusted inputs/outputs. Moreover, we assume a strong attacker observing execution time and control flow, following the leakage model of *constant-time security* [4]. Finally, we assume the attacker observes the read locations (e.g., via probing the cache)—another usual assumption of constant-time security. We thus define $\mathsf{isInt}(s) = \neg\,\mathsf{final}(s)$ and $\mathsf{getInt}(s) = (\mathsf{getAct}(s), \mathsf{getObs}(s))$ where, if $s = (ps, cfg_0 \cdot \ldots \cdot cfg_k, inp, L)$, we have:

$$\mathsf{getAct}(s) = \begin{cases} \mathsf{head}(inp_\mathsf{U}), & \text{if } k = 0 \text{ and } \mathsf{isInput}_\mathsf{U}(cfg_0) \\ \bot, & \text{otherwise} \end{cases}$$

$$\mathsf{getObs}(s) = \begin{cases} (\mathsf{outOf}(cfg_0), L), & \text{if } k = 0 \text{ and} \\ & \quad \mathsf{isOutput}_\mathsf{U}(cfg_0) \\ \bot, & \text{otherwise} \end{cases}$$

As shown by the definition of isInt, interaction happens everywhere except for the final states (where the system is idle)—this pervasiveness of interaction means that the attacker can observe the execution time. For most interaction points, getInt (which pairs actions via getAct with observations via getObs) reveals nothing beyond the fact that an execution step has been taken, i.e., returns $(\bot, \bot)$; exceptions are when the current command is an untrusted input or output command (which we expressed by the $\mathsf{isInput}_\mathsf{U}$ and $\mathsf{isOutput}_\mathsf{U}$ predicates). Note also that inputs and outputs can only take place non-speculatively, i.e., at speculation level 0.

The second question, which will determine isSec and getSec, is what the sensitive data, i.e., the secrets, are. For the bulk of our examples, fun1–fun5, we take $\mathsf{isSec}(s)$ to say that s is initial and $\mathsf{getSec}(s)$ to return the memory part of this initial state (namely μ_0). In short, here the entire initial memory constitutes the secrets.

For fun6, we change isSec and getSec to account for the interactive nature of the secrets, entered in the system as trusted inputs and produced as trusted outputs. Thus, isSec now says that the state is not final and not speculating and, for a state $s = (ps, cfgs, inp, L)$, $\mathsf{getSec}(s)$ now returns triples (A, B, C) where:

- A is (as before) the memory part of s provided s is initial, and $\bot$ otherwise;
- B is $\mathsf{head}(inp_\mathsf{T})$ if the command of $cfgs_0$ is an $\mathsf{Input}_\mathsf{T}$ (trusted input) command, and $\bot$ otherwise;
- C is either the (trusted) output produced if the command of $cfgs_0$ is an $\mathsf{Output}_\mathsf{T}$ one, and $\bot$ otherwise.

In short, the secrets are now three-fold: memory if in initial state, trusted input if any, and trusted output if any.

With this instantiation, one can check that the intuitive (in)security of these examples discussed in §2.1 corresponds to their (not) satisfying relative security. For example, fun1 is not relatively secure essentially because the attacker can use the $\mathsf{Input}_\mathsf{U}\ x$ command to pass an input $x \geq N$, which leaks $a[x]$ (part of the initial memory) via the read-locations component of the observation (L). Indeed, the value of $a[x]$ is collected by the semantics among the read locations while executing line 4 in the speculative semantics at speculation level 1; but not under normal semantics, under which line 4 is not executed because $x < N$. This violation of relative security is impossible for fun2 and fun3, where the fences stop speculation via the FENCE rule in Fig. 4 before the location $a[x]$ can be collected. In fact, fun2–fun6 are all relatively secure, like our informal analysis in §2.1 concluded. But actual proofs for these will be based on the unwinding proof method we introduce in §5, and will be sketched in §6.

4. Connections with notions from the literature

Of the vast literature on Spectre and Meltdown [3], we will only discuss formal modeling and verification aspects.

The contract/policy pattern. Most of the approaches focus on protecting the secrecy of (aspects of) the initial state, expressed as an indistinguishability relation $\simeq$ on State, and on explicitly modeling only the attacker's observations (our function O) and not the attacker's actions (our function

```
0 :   Start ;
1 :   Input_U x ;
2 :   t = 0 ;
3 :   IfJump (x < N) 4 5 ;
4 :       t = b[a[x] * 512] ;
5 :   Output_U t
```

fun1

```
0 :   Start ;
1 :   Input_U x ;
2 :   t = 0 ;
3 :   IfJump (x < N) 4 6 ;
4 :       Fence ;
5 :       t = b[a[x] * 512] ;
6 :   Output_U t
```

fun2

```
0 :   Start ;
1 :   Input_U x ;
2 :   t = 0 ;
3 :   IfJump (x < N) 4 7 ;
4 :       v = a[x] ;
5 :       Fence ;
6 :       t = b[v * 512] ;
7 :   Output_U t
```

fun3

```
0 :   Start ;
1 :   Input_U x ;
2 :   t = 0 ;
3 :   IfJump (x < N) 4 6 ;
4 :       v = a[0] ;
5 :       t = b[v * 512] ;
6 :   Output_U t
```

fun4

```
 0 :   Start ;
 1 :   t = 0 ;
 2 :   x = 1 ;
 3 :   IfJump (not (x == 0)) 4 11 ;
 4 :       Input_U x ;
 5 :       IfJump (x < N) 6 10 ;
 6 :           v = a[x] ;
 7 :           Fence ;
 8 :           t = b[v * 512] ;
 9 :       Output_U t ;
10 :   Jump 3 ;
11 :   Output_U 0 ;
```

fun5

```
 0 :   Start ;
 1 :   t = 0 ;
 2 :   x = 1 ;
 3 :   IfJump (not (x == 0)) 4 13 ;
 4 :       Input x ;
 5 :       Input_T y ;
 6 :       IfJump (x < N) 7 12 ;
 7 :           v = a[x] ;
 8 :           Output_T (F(x, y)) ;
 9 :           Fence ;
10 :           t = b[v * 512] ;
11 :       Output_U t ;
12 :   Jump 3 ;
13 :   Output_U 0 ;
```

fun6

Fig. 5: Our six motivating examples from §2.1 written in IMP

A). Notably, this is the case of Guanciale et al.'s *conditional non-interference* [12], an extension of noninterference [10, 27] used to build a detailed formal model of Spectre vulnerabilities in the presence of speculative and out-of-order execution; and of Guarnieri et al.'s *speculative non-interference* [13] (which forms the basis of Spectator, an automatic tool for proving security against Spectre).

These and several other approaches are surveyed in a recent SoK paper by Cauligi et al. [4], which, borrowing concepts and terminology from Guarnieri et al. [14], describes these approaches uniformly under the following *contract/policy* pattern, characterized by three parameters: **1)** an *execution model* α, indicating the states explored during executions, **2)** a *leakage model* l, indicating the observations that are possible along executions, and **3)** a *(secrecy) policy* p, indicating the secrets stored in the initial state, via an indistinguishability relation on states $\simeq_p$.

An execution model α corresponds to an operational semantics (our system models $\mathcal{SM} = (\mathsf{State}, \mathsf{istate}, \Rightarrow)$). The combination between a leakage and an execution model, called a *contract*, determines a state-observation function $[\![\cdot]\!]_l^\alpha : \mathsf{State} \to \mathsf{Seq}(\mathsf{Obs})$. Given execution and leakage models α and l and a policy p, direct noninterference states that the observation function cannot detect any secret, in that $\forall s_1, s_2 \in \mathsf{State}.\ s_1 \simeq_p s_2 \longrightarrow [\![s_1]\!]_l^\alpha = [\![s_2]\!]_l^\alpha$. Relative noninterference generalizes this to two contracts, namely (in our terminology) a vanilla one (α, l) and an optimization-enhanced one (α', l'), and a common policy p, stating that any secrets that are leaked by (α', l') are also leaked by (α, l), in that $\forall s_1, s_2 \in \mathsf{State}.\ s_1 \simeq_p s_2 \wedge [\![s_1]\!]_{l'}^{\alpha'} = [\![s_2]\!]_{l'}^{\alpha'} \longrightarrow [\![s_1]\!]_l^\alpha = [\![s_2]\!]_l^\alpha$.

Assuming that the state-observation function stems from a trace observation function (which seems true in all interesting cases), the contract/policy pattern is an instance of our relative security by instantiating our attacker models: $\mathsf{S}(\pi)$ as a singleton sequence, namely the $\simeq_p$-equivalence class of the trace's starting state, $\mathsf{O}(\pi)$ as the trace observation function underlying $[\![\cdot]\!]_l^\alpha$, and $\mathsf{A}(\pi)$ as the empty sequence.

The attacker actions A playing no role in the above models and the secrets being restricted to the initial state means that such models, while easily capturing examples like the ones in our Listings 1–4, are not directly suitable for more interactive examples as in Listings 5 and 6. While interaction features are not completely out of scope for these models, we believe our approach to allow interaction natively in the abstract models can simplify reasoning.

TPOD. Cheang et al.'s *TPOD (trace-property dependent observational nondeterminism)* [5] is an exception to the above rule, in that it explicitly captures both active attackers *and* interactive uploading of the secrets. TPOD is an extension of observational determinism [33] from a two-trace to a four-trace property. It is parameterized by a notion of low-equivalence $\equiv_{\mathsf{low}}$ on states, which describes what an attacker cannot distinguish, and by low and high operations taken at each transition between states: $\mathsf{op}_{\mathsf{low}}(s)$ and $\mathsf{op}_{\mathsf{high}}(s)$ are the high and low operations applied when transiting from state s to the next state. Low equivalence and the high and low operations are extended from states to traces componentwise. Finally, rather than considering two distinct system models, (in our terminology) a vanilla one and an optimization-enhanced one, TPOD uses a single system for both and instead uses a set T of (what we call)

vtraces. With these parameters fixed, TPOD is expressed as follows (where π_i^0 denotes the starting state of π_i):

$$\forall \pi_1, \pi_2 \in \mathsf{Trace} \setminus T.\ \forall \hat{\pi}_1, \hat{\pi}_2 \in T.$$
$$\mathsf{op}_{\mathsf{low}}(\hat{\pi}_1) = \mathsf{op}_{\mathsf{low}}(\hat{\pi}_2) = \mathsf{op}_{\mathsf{low}}(\pi_2) = \mathsf{op}_{\mathsf{low}}(\pi_1)\ \wedge$$
$$\mathsf{op}_{\mathsf{high}}(\hat{\pi}_1) = \mathsf{op}_{\mathsf{high}}(\pi_1)\ \wedge\ \mathsf{op}_{\mathsf{high}}(\hat{\pi}_2) = \mathsf{op}_{\mathsf{high}}(\pi_2)\ \wedge$$
$$\hat{\pi}_1 \equiv_{\mathsf{low}} \hat{\pi}_2\ \wedge\ \pi_1^0 \equiv_{\mathsf{low}} \pi_2^0$$
$$\longrightarrow \pi_1 \equiv_{\mathsf{low}} \pi_2$$

We can parse TPOD in terms of our relative security ingredients: **1)** the sequence of low-equivalence classes $s_0/_{\equiv_{\mathsf{low}}} s_1/_{\equiv_{\mathsf{low}}} \ldots$ of a trace $\pi = s_0\, s_1 \ldots$ form the observations $\mathsf{O}(\pi)$; **2)** the low operations $\mathsf{op}_{\mathsf{low}}(\pi)$ form the actions $\mathsf{A}(\pi)$; **3)** the high operations $\mathsf{op}_{\mathsf{high}}(\pi)$ form the secrets $\mathsf{S}(\pi)$.

However, TPOD is *not* a particular case of our relative security because of two reasons, highlighted in the above formula. First, as shown by the highlighted equality, the TPOD formula constrains the vtraces $\hat{\pi}_1$ and $\hat{\pi}_2$ to have the same low operations (in our terminology, the same actions) not only with each other, but also with their counterpart otraces π_1 and π_2; this (over)constrains any leak exhibited by the optimization-enhanced system to be reproduced by the vtraces under the same attacker actions, which in our opinion is not justified. Indeed, our design of relative security suggests that the similarities between the vtraces and the otraces should refer to the underlying secrets, not to the attacker-taken actions; this is because the standard assumption is that the attacker is free to take *any* actions to exhibit a leak. Interestingly, this overconstraining problem with TPOD is illustrated by an example in the TPOD paper itself [5], namely the conditionally vulnerable program we showed in Listing 4 (Fig. 3(c) in [5]): Assuming $N > 0$, under the interpretation of low actions as the inputs to the function `fun4` (which is natural, and is the one endorsed in [5]), TPOD requires that the vtraces reproduce the leak with the same input—which is impossible because, as we explained in §2.1 (following a discussion from [5]), reproducing the leak in the vanilla semantics requires a specific input smaller than N which is different from the one in the otraces, e.g., if $N = 2$ then we need the vanilla input to be 0 or 1. Thus, contrary to the authors' suggestion, Listing 4's example does *not* satisfy TPOD; though it satisfies relative security.

The other difference between relative security and TPOD is shown in the highlighted quantifier: Because it quantifies universally rather than existentially over the vtraces $\hat{\pi}_1$ and $\hat{\pi}_2$, the TPOD formula asks that any leak of π_1 and π_2 is reproduced not just by *some* pair of vtraces $\hat{\pi}_1$ and $\hat{\pi}_2$ (which seems natural), but rather, more demandingly, by *all* pairs of vtraces that happen to have the same secrets with π_1 and π_2.

In summary, TPOD is a strong property that mischaracterizes intuitively secure programs that our relative security characterizes correctly. However, it is the TPOD strength that enables successful automatic verification which deems secure several interesting programs (as elaborated in [5]).

5. (Dis)Proof Methods for Relative Security

This section develops incremental proof and disproof methods for relative security, which can be enabled provided the secrets, observations and actions of attacker models are themselves defined incrementally on traces (§2.5). After discussing design challenges stemming from the four-trace constraint system specific to relative security (§5.1), we converge to a definition of unwinding relation (§5.2); we distinguish between the finite-trace case and the general case, the former allowing for simpler conditions. Our discussion culminates with the formal statement that unwinding indeed ensures relative security (§5.3). We also introduce a compositional variant of unwinding (§5.4). Finally, we look into the dual problem, of incrementally *disproving* relative security, which leads to a proof method that we call secret-directed unwinding because it is a form of unwinding that shows the impossibility of saturating given sequences of secrets (§5.5).

We fix two system models $\mathcal{SM}_{\mathsf{van}} = (\mathsf{State}_{\mathsf{van}}, \mathsf{istate}_{\mathsf{van}}, \Rightarrow_{\mathsf{van}})$ and $\mathcal{SM}_{\mathsf{opt}} = (\mathsf{State}_{\mathsf{opt}}, \mathsf{istate}_{\mathsf{opt}}, \Rightarrow_{\mathsf{opt}})$ such that $\mathsf{State}_{\mathsf{van}} \subseteq \mathsf{State}_{\mathsf{opt}}$, and state-wise attacker models for them, $\mathcal{AM}_{\mathsf{van}} = (\mathsf{Sec}, \mathsf{S}_{\mathsf{van}}, \mathsf{Obs}_{\mathsf{van}}, \mathsf{O}_{\mathsf{van}}, \mathsf{Act}_{\mathsf{van}}, \mathsf{A}_{\mathsf{van}})$ and $\mathcal{AM}_{\mathsf{opt}} = (\mathsf{Sec}, \mathsf{S}_{\mathsf{opt}}, \mathsf{Obs}_{\mathsf{opt}}, \mathsf{O}_{\mathsf{opt}}, \mathsf{Act}_{\mathsf{opt}}, \mathsf{A}_{\mathsf{opt}})$ (having the same set of secrets Sec). Recall from §2.5 that the attacker models being state-wise means the existence of the predicates $\mathsf{isSec}_u : \mathsf{State}_u \to \mathsf{Bool}$, $\mathsf{getSec}_u : \mathsf{State}_u \to \mathsf{Sec}$, $\mathsf{isInt}_u : \mathsf{State}_u \to \mathsf{Bool}$, $\mathsf{getObs}_u : \mathsf{State}_u \to \mathsf{Obs}_u$, and $\mathsf{getAct}_u : \mathsf{State}_u \to \mathsf{Act}_u$ that define the functions S_u, O_u and A_u state-wise, where $u \in \{\mathsf{opt}, \mathsf{van}\}$.

We will describe methods for (dis)proving $(\mathcal{SM}_{\mathsf{opt}}, \mathcal{AM}_{\mathsf{opt}}) \geq_{\checkmark} (\mathcal{SM}_{\mathsf{van}}, \mathcal{AM}_{\mathsf{van}})$ and $(\mathcal{SM}_{\mathsf{opt}}, \mathcal{AM}_{\mathsf{opt}}) \geq_{\checkmark}^{\mathsf{fin}} (\mathcal{SM}_{\mathsf{van}}, \mathcal{AM}_{\mathsf{van}})$. We will only use the subscripts van and opt for the state-set components, e.g., $\mathsf{State}_{\mathsf{van}}$ and $\mathsf{State}_{\mathsf{opt}}$, and omit them for the other componentsm, e.g., $\Rightarrow$.

5.1. Design aspects

Unwinding is a (bi)simulation-like [30] method specialized in proving noninterference and related two-trace properties [11, 20]. It exhibits a winning strategy for a two-player game that incrementally follows a trace π_1 controlled by an antagonist, while constructing a similar trace π_2 controlled by a protagonist while "countering" any possible leak.

For relative security, there are additional challenges when designing an unwinding-like proof method since we must cope with not two but four traces $\pi_1, \pi_2, \hat{\pi}_1, \hat{\pi}_2$, as depicted in Fig. 1. Here, the otraces π_1 and π_2 on the one hand, and also their counterpart vtraces $\hat{\pi}_1$ and $\hat{\pi}_2$ on the other, both pairs correspond to the pair of traces from traditional unwinding. However, there are different requirements on π_1, π_2 (which are *allowed* to exhibit a leak) and $\hat{\pi}_1, \hat{\pi}_2$ (which must *reproduce* the leak exhibited by π_1, π_2).

We are after a mechanism for building the vtraces $\hat{\pi}_1$ and $\hat{\pi}_2$ incrementally from the otraces π_1 and π_2 in such a way that they create the same leak, i.e., take the same actions *and* generate the same secrets *yet* produce different observations. During the build process, we will have to simultaneously maintain the following relationships between these traces:

- (R1) the otraces π_1 and π_2 have the same actions but different observations;
- (R2) the vtraces $\hat{\pi}_1$ and $\hat{\pi}_2$ have the same actions but different observations;

(R3) the otrace π_i and its vtrace counterpart $\hat{\pi}_i$ have the same secrets for $i \in \{1, 2\}$.

We must also factor in the polarities of these relationships. First, as the traces evolve during the unwinding game, relationship (R1) will be *assumed*, whereas relationships (R2) and (R3) must be *guaranteed*. Moreover, assuming/guaranteeing that two generated sequences *stay equal* (like for the sequences of actions in relationships (R1)–(R3)) has a different flavour from assuming/guaranteeing that two generated sequences *become different* (like for the sequences of observations in relationships (R1) and (R2))—roughly, we are talking about a safety versus a liveness property.

To capture these nuances, we will maintain the status of the observations' divergence at (R1) and (R2), by remembering whether π_1 and π_2 have (already) differed in their observations (meaning the status is "diff") or not (yet), meaning the status is "eq" (and similarly for $\hat{\pi}_1$ and $\hat{\pi}_2$). Thus, we can guarantee (R2) when assuming (R1) by:

- starting with the status of π_1 vs. π_2, as well as that of $\hat{\pi}_1$ vs. $\hat{\pi}_2$, set to eq;
- making sure the status of $\hat{\pi}_1$ vs. $\hat{\pi}_2$ changes (from eq to diff) as soon as the status for π_1 vs. π_2 changes.

The vtraces $\hat{\pi}_1$ and $\hat{\pi}_2$ will grow as π_1 and π_2 grow, using one of the following transition-matching mechanisms:

- $\hat{\pi}_i$ takes a step to match a step by π_i; or
- both $\hat{\pi}_1$ and $\hat{\pi}_2$ take a synchronized step to match a synchronized step by π_1 and π_2; or
- $\hat{\pi}_1$ and/or $\hat{\pi}_2$ ignore the step taken either separately or synchronously by π_1 and/or π_2.

These matching patterns are the most natural choices for the reactive growth of $\hat{\pi}_i$'s based on that of the π_i's. But for extra flexibility in proofs we will also allow variations of these patterns—e.g., $\hat{\pi}_1$ alone taking a step in response to a synchronized step by π_1 and π_2, or conversely $\hat{\pi}_1$ and $\hat{\pi}_2$ taking a synchronized step in response to a step by π_1. The side-conditions when applying these matching mechanisms ensure that we maintain "(R1) implies (R2) and (R3)".

In what follows, we will refer to the conditions (R1)–(R3) more casually and intuitively, as:

- an *interaction contract*, corresponding to "(R1) implies (R2)", about the actions of $\hat{\pi}_1$ and $\hat{\pi}_2$ being the same and the observations being (eventually) different (the latter provided this is the case for π_1 and π_2);
- a *secrecy contract*, corresponding to "((R1) implies) (R2)", about the produced secrets being the same between $\hat{\pi}_i$ and π_i.

In addition to the above discussed *reactive* growth of the $\hat{\pi}_i$'s, we will also allow their *proactive* growth. This will ensure further flexibility in proofs by enabling the $\hat{\pi}_i$'s to take "independent" moves, i.e., moves not triggered by moves of the π_i's. However, to ensure soundness we will need to restrict proactive growth using *timers* (as we will explain).

5.2. Definition of unwinding

Below is the formal definition. We let $\mathsf{Status} = \{\mathsf{eq}, \mathsf{diff}\}$, where eq signifies equality and diff difference/ divergence, and $\mathbb{N}_\infty = \mathbb{N} \cup \{\infty\}$, the set of natural numbers extended with ∞—these will be our timer parameters. We let $\mathsf{SState}_u \doteq \mathsf{State}_u \times \mathsf{State}_u \times \mathsf{Status}$, where $u \in \{\mathsf{opt}, \mathsf{van}\}$.

Def. 3 A relation $\Delta : \mathbb{N}_\infty \to (\mathbb{N}_\infty \times \mathbb{N}_\infty) \to \mathsf{SState}_{\mathsf{opt}} \to \mathsf{SState}_{\mathsf{van}} \to \mathsf{Bool}$ is an *unwinding* when, for all $v \in \mathbb{N}_\infty$, $v_1, v_2 \in \mathbb{N}_\infty$, $s_1, s_2 \in \mathsf{State}_{\mathsf{opt}}$, $\hat{s}_1, \hat{s}_2 \in \mathsf{State}_{\mathsf{van}}$ and $st, \hat{st} \in \mathsf{Status}$, if $\Delta\ v\ (v_1, v_2)(s_1, s_2, st)\ (\hat{s}_1, \hat{s}_2, \hat{st})$ then:

- $st = \mathsf{eq}$ implies $\mathsf{isInt}(s_1) \leftrightarrow \mathsf{isInt}(s_2)$;
- $\mathsf{final}(s_1) \leftrightarrow \mathsf{final}(s_2) \leftrightarrow \mathsf{final}(\hat{s}_1) \leftrightarrow \mathsf{final}(\hat{s}_2)$;
- either $\mathsf{react}(\Delta)\ (v_1, v_2)(s_1, s_2, st)\ (\hat{s}_1, \hat{s}_2, \hat{st})$, or $\exists w < v.\ \mathsf{proact}(\Delta)\ w\ (v_1, v_2)(s_1, s_2, st)\ (\hat{s}_1, \hat{s}_2, \hat{st})$.

The predicates $\mathsf{react}(\Delta)$ and $\mathsf{proact}(\Delta)$, covering the aforementioned reactive and proactive components of unwinding, are defined in Figs. 6 and 7, respectively. The definitions in these figures use the parameters of relative security either directly (e.g., isSec and isInt) or via the following auxiliary functions newStat, eqSec and eqAct:

$$
\begin{aligned}
\mathsf{eqSec}(s, s') &\doteq (\mathsf{isSec}(s) \leftrightarrow \mathsf{isSec}(s')) \wedge \\
&\quad\ (\mathsf{isSec}(s) \longrightarrow \mathsf{getSec}(s) = \mathsf{getSec}(s')) \\
\mathsf{eqAct}(s, s') &\doteq (\mathsf{isInt}(s) \leftrightarrow \mathsf{isInt}(s')) \wedge \\
&\quad\ (\mathsf{isInt}(s) \longrightarrow \mathsf{getAct}(s) = \mathsf{getAct}(s')) \\
\mathsf{newStat}(st, s, s') &\doteq \begin{cases} \mathsf{diff} & \text{if } \mathsf{isInt}(s) \wedge \mathsf{isInt}(s') \wedge \\ & \quad \mathsf{getObs}(s) \neq \mathsf{getObs}(s') \\ st & \text{otherwise} \end{cases}
\end{aligned}
$$

Thus, the predicate $\mathsf{eqSec}(s, s')$ holds when the states s and s' have equal secrets, if any; and similarly for eqAct concerning actions. The function $\mathsf{newStat}(st, s, s')$ tracks any change to the status st, from eq to diff, that may have occurred due to the observations in s and s'.

Def. 4 A relation $\Delta : \mathbb{N}_\infty \to \mathsf{SState}_{\mathsf{opt}} \to \mathsf{SState}_{\mathsf{van}} \to \mathsf{Bool}$ is said to be a *finitary unwinding* if the condition from Def. 3 holds when ignoring the two $\mathbb{N}_\infty$ arguments highlighted in gray (including in Figs. 6 and 7).

Discussion on finitary unwinding. We first focus on finitary unwinding (Def. 4), ignoring everything highlighted in gray—this will be enough for finitary relative security. A finitary unwinding relation Δ encodes the current states of the otraces, s_1 and s_2, and their observation divergence status st (call them "ostates" and "ostatus"), along with the current states $\hat{s}_1$ and $\hat{s}_2$ and status $\hat{st}$ of the vtraces which are being constructed ("vstates" and "vstatus"), together with a timer parameter $v \in \mathbb{N}_\infty$ (for bounding proactive growth, explained later) expressed as $\Delta\ v\ (s_1, s_2, st)\ (\hat{s}_1, \hat{s}_2, \hat{st})$. Def. 4 ensures that Δ can support the secure growth of the vtraces, reactively or proactively.

A note on notation: In Figs. 6 and 7, we use superscripts to indicate which subset of the two ostates take transitions, and subscripts to indicate which of the two vstates are reacting. For example, $\mathsf{match}_2^{1,2}(\Delta)$ means that s_1 and s_2 both take transitions, and $\hat{s}_2$ takes a matching transition in reaction. Superscripts refer to *demonic nondeterminism*: we must consider all three cases; and indeed, $\mathsf{react}(\Delta)$ is defined as the *conjunction* of $\mathsf{match}^1(\Delta)$, $\mathsf{match}^2(\Delta)$ and

$$\mathsf{react}(\Delta)\,(v_1,v_2)(s_1,s_2,st)\;(\hat{s}_1,\hat{s}_2,\hat{st}) \;\doteq\; \mathsf{match}^1(\Delta)\,(v_1,v_2)(s_1,s_2,st)\;(\hat{s}_1,\hat{s}_2,\hat{st})\wedge$$
$$\mathsf{match}^2(\Delta)\,(v_1,v_2)(s_1,s_2,st)\;(\hat{s}_1,\hat{s}_2,\hat{st})\wedge$$
$$\mathsf{match}^{1,2}(\Delta)\,(v_1,v_2)(s_1,s_2,st)\;(\hat{s}_1,\hat{s}_2,\hat{st})$$

$$\mathsf{match}^1(\Delta)\,(v_1,v_2)(s_1,s_2,st)\;(\hat{s}_1,\hat{s}_2,\hat{st}) \;\doteq$$
$$\neg\,\mathsf{isInt}(s_1)\;\longrightarrow$$
$$(\forall s_1'.\; s_1 \Rightarrow s_1' \;\longrightarrow$$
$$(\exists (w_1,w_2)<(v_1,v_2).\;\neg\,\mathsf{isSec}(s_1)\;\wedge\;\Delta\,\infty\,(w_1,w_2)(s_1',s_2,st)\;(\hat{s}_1,\hat{s}_2,\hat{st}))\;\vee$$
$$(\exists w_2<v_2.\;\mathsf{eqSec}(s_1,\hat{s}_1)\;\wedge\;\neg\,\mathsf{isInt}(\hat{s}_1)\;\wedge\;\mathsf{match}^1_1(\Delta)\;(\infty,w_2)(s_1',s_2,st)\;(\hat{s}_1,\hat{s}_2,\hat{st}))\;\vee$$
$$(\mathsf{eqSec}(s_1,\hat{s}_1)\;\wedge\;\mathsf{eqAct}(\hat{s}_1,\hat{s}_2)\;\wedge\;\neg\,\mathsf{isSec}(\hat{s}_2)\;\wedge\;\mathsf{match}^1_{1,2}(\Delta)(\infty,\infty)\;(s_1',s_2,st)\;(\hat{s}_1,\hat{s}_2,\hat{st})))$$

$$\mathsf{match}^2(\Delta)\,(v_1,v_2)(s_1,s_2,st)\;(\hat{s}_1,\hat{s}_2,\hat{st}) \;\doteq$$
$$\neg\,\mathsf{isInt}(s_2)\;\longrightarrow$$
$$(\forall s_2'.\; s_2 \Rightarrow s_2' \;\longrightarrow$$
$$(\exists (w_1,w_2)<(v_1,v_2).\;\neg\,\mathsf{isSec}\;s_2\;\wedge\;\Delta\,\infty\,(w_1,w_2)(s_1,s_2',st)\;(\hat{s}_1,\hat{s}_2,\hat{st}))\;\vee$$
$$(\exists w_1<v_1.\;\mathsf{eqSec}(s_2,\hat{s}_2)\;\wedge\;\neg\,\mathsf{isInt}(\hat{s}_2)\;\wedge\;\mathsf{match}^2_2(\Delta)\;(w_1,\infty)(s_1,s_2',st)\;(\hat{s}_1,\hat{s}_2,\hat{st}))\;\vee$$
$$(\mathsf{eqSec}(s_2,\hat{s}_2)\;\wedge\;\mathsf{eqAct}(\hat{s}_1,\hat{s}_2)\;\wedge\;\neg\,\mathsf{isSec}\;\hat{s}_1\;\wedge\;\mathsf{match}^2_{1,2}(\Delta)\;(\infty,\infty)(s_1,s_2',st)\;(\hat{s}_1,\hat{s}_2,\hat{st})))$$

$$\mathsf{match}^{1,2}(\Delta)\,(v_1,v_2)(s_1,s_2,st)\;(\hat{s}_1,\hat{s}_2,\hat{st}) \;\doteq$$
$$\text{let } st' = \mathsf{newStat}(st,s_1,s_2) \text{ in}$$
$$\mathsf{isInt}(s_1)\;\wedge\;\mathsf{isInt}(s_2)\;\wedge\;\mathsf{eqAct}(s_1,s_2)\;\longrightarrow$$
$$(\forall s_1',s_2'.\; s_1 \Rightarrow s_1'\;\wedge\; s_2\Rightarrow s_2' \;\longrightarrow$$
$$(\exists (w_1,w_2)<(v_1,v_2).\;\neg\,\mathsf{isSec}(s_1)\;\wedge\;\neg\,\mathsf{isSec}(s_2)\;\wedge\;(st'=st\;\vee\;\hat{st}=\mathsf{diff})\;\wedge\;\Delta\,\infty\;(w_1,w_2)(s_1',s_2',st')\;(\hat{s}_1,\hat{s}_2,\hat{st}))\;\vee$$
$$(\exists w_2<v_2.\;\mathsf{eqSec}(s_1,\hat{s}_1)\;\wedge\;\neg\,\mathsf{isSec}(s_2)\;\wedge\;\neg\,\mathsf{isInt}(\hat{s}_1)\;\wedge\;(st'=st\;\vee\;\hat{st}=\mathsf{diff})\;\wedge\;\mathsf{match}^{1,2}_1(\Delta)\;(\infty,w_2)(s_1',s_2',st')\;(\hat{s}_1,\hat{s}_2,\hat{st}))\;\vee$$
$$(\exists w_1<v_1.\;\neg\,\mathsf{isSec}(s_1)\;\wedge\;\mathsf{eqSec}(s_2,\hat{s}_2)\;\wedge\;\neg\,\mathsf{isInt}(\hat{s}_2)\;\wedge\;(st'=st\;\vee\;\hat{st}=\mathsf{diff})\;\wedge\;\mathsf{match}^{1,2}_2(\Delta)\;(w_1,\infty)(s_1',s_2',st')\;(\hat{s}_1,\hat{s}_2,\hat{st}))\;\vee$$
$$(\mathsf{eqSec}(s_1,\hat{s}_1)\;\wedge\;\mathsf{eqSec}(s_2,\hat{s}_2)\;\wedge\;\mathsf{eqAct}(\hat{s}_1,\hat{s}_2)\;\wedge\;\mathsf{match}^{1,2}_{1,2}(\Delta)\;(\infty,\infty)(s_1',s_2',st')\;(\hat{s}_1,\hat{s}_2,\hat{st}))$$

$$\mathsf{match}^1_1(\Delta)\,(v_1,v_2)(s_1',s_2,st)\;(\hat{s}_1,\hat{s}_2,\hat{st}) \;\doteq\; \exists \hat{s}_1'\in\mathsf{State}_{\mathsf{van}}.\;\hat{s}_1\Rightarrow\hat{s}_1'\;\wedge\;\Delta\,\infty\,(v_1,v_2)(s_1',s_2,st)\;(\hat{s}_1',\hat{s}_2,\hat{st})$$

$$\mathsf{match}^1_{1,2}(\Delta)\,(v_1,v_2)(s_1',s_2,st)\;(\hat{s}_1,\hat{s}_2,\hat{st}) \;\doteq\; \exists \hat{s}_1',\hat{s}_2'\in\mathsf{State}_{\mathsf{van}}.\;\hat{s}_1\Rightarrow\hat{s}_1'\;\wedge\;\hat{s}_2\Rightarrow\hat{s}_2'\;\wedge\;\Delta\,\infty\,(v_1,v_2)(s_1',s_2,st)\;(\hat{s}_1',\hat{s}_2',\hat{st})$$

$$\mathsf{match}^2_2(\Delta)\,(v_1,v_2)(s_1,s_2',st)\;(\hat{s}_1,\hat{s}_2,\hat{st}) \;\doteq\; \exists \hat{s}_2'\in\mathsf{State}_{\mathsf{van}}.\;\hat{s}_2\Rightarrow\hat{s}_2'\;\wedge\;\Delta\,\infty\,(v_1,v_2)(s_1,s_2',st)\;(\hat{s}_1,\hat{s}_2',\hat{st})$$

$$\mathsf{match}^2_{1,2}(\Delta)\,(v_1,v_2)(s_1,s_2',st)\;(\hat{s}_1,\hat{s}_2,\hat{st}) \;\doteq\; \exists \hat{s}_1',\hat{s}_2'\in\mathsf{State}_{\mathsf{van}}.\;\hat{s}_1\Rightarrow\hat{s}_1'\;\wedge\;\hat{s}_2\Rightarrow\hat{s}_2'\;\wedge\;\Delta\,\infty\,(v_1,v_2)(s_1,s_2',st)\;(\hat{s}_1',\hat{s}_2',\hat{st})$$

$$\mathsf{match}^{1,2}_1(\Delta)\,(v_1,v_2)(s_1',s_2',st')\;(\hat{s}_1,\hat{s}_2,\hat{st}) \;\doteq\; \exists \hat{s}_1'\in\mathsf{State}_{\mathsf{van}}.\;\hat{s}_1\Rightarrow\hat{s}_1'\;\wedge\;\Delta\,\infty\,(v_1,v_2)(s_1',s_2',st')\;(\hat{s}_1',\hat{s}_2,\hat{st})$$

$$\mathsf{match}^{1,2}_2(\Delta)\,(v_1,v_2)(s_1',s_2',st')\;(\hat{s}_1,\hat{s}_2,\hat{st}) \;\doteq\; \exists \hat{s}_2'\in\mathsf{State}_{\mathsf{van}}.\;\hat{s}_2\Rightarrow\hat{s}_2'\;\wedge\;\Delta\,\infty\,(v_1,v_2)(s_1',s_2',st')\;(\hat{s}_1,\hat{s}_2',\hat{st})$$

$$\mathsf{match}^{1,2}_{1,2}(\Delta)\,(v_1,v_2)(s_1',s_2',st')\;(\hat{s}_1,\hat{s}_2,\hat{st}) \;\doteq$$
$$\text{let } \hat{st}' = \mathsf{newStat}(\hat{st},\hat{s}_1,\hat{s}_2) \text{ in}$$
$$\exists \hat{s}_1',\hat{s}_2'\in\mathsf{State}_{\mathsf{van}}.\;\hat{s}_1\Rightarrow\hat{s}_1'\;\wedge\;\hat{s}_2\Rightarrow\hat{s}_2'\;\wedge\;(st'=\mathsf{diff}\;\longrightarrow\;\hat{st}'=\mathsf{diff})\;\wedge\;\Delta\,\infty\,(v_1,v_2)(s_1',s_2',st')\;(\hat{s}_1',\hat{s}_2,\hat{st}')$$

Fig. 6: Definition of $\mathsf{react}(\Delta)$ – the reactive component of unwinding

$$\mathsf{proact}(\Delta)\;v\,(v_1,v_2)(s_1,s_2,st)\;(\hat{s}_1,\hat{s}_2,\hat{st}) \;\doteq$$
$$\neg\,\mathsf{isSec}(s_1)\;\wedge\;\neg\,\mathsf{isInt}(s_1)\;\wedge\;\mathsf{imove}^1(\Delta)\;v\,(v_1,v_2)(s_1,s_2,st)\;(\hat{s}_1,\hat{s}_2,\hat{st})\;\vee$$
$$\neg\,\mathsf{isSec}(s_1)\;\wedge\;\neg\,\mathsf{isInt}(s_1)\;\wedge\;\mathsf{imove}^2(\Delta)\;v\,(v_1,v_2)(s_1,s_2,st)\;(\hat{s}_1,\hat{s}_2,\hat{st})\;\vee$$
$$\neg\,\mathsf{isSec}(s_1)\;\wedge\;\neg\,\mathsf{isSec}(s_2)\;\wedge\;\mathsf{eqAct}(s_1,s_2)\;\wedge\;\mathsf{imove}^{1,2}(\Delta)\;v\,(v_1,v_2)(s_1,s_2,st)\;(\hat{s}_1,\hat{s}_2,\hat{st})$$

$$\mathsf{imove}_1(\Delta)\;v\,(v_1,v_2)(s_1,s_2,st)\;(\hat{s}_1,\hat{s}_2,\hat{st}) \;\doteq\; \exists \hat{s}_1'\in\mathsf{State}_{\mathsf{van}}.\;\hat{s}_1\Rightarrow\hat{s}_1'\;\wedge\;\Delta\;v\,(v_1,v_2)(s_1,s_2,st)\;(\hat{s}_1',\hat{s}_2,\hat{st})$$

$$\mathsf{imove}_2(\Delta)\;v\,(v_1,v_2)(s_1,s_2,st)\;(\hat{s}_1,\hat{s}_2,\hat{st}) \;\doteq\; \exists \hat{s}_2'\in\mathsf{State}_{\mathsf{van}}.\;\hat{s}_2\Rightarrow\hat{s}_2'\;\wedge\;\Delta\;v\,(v_1,v_2)(s_1,s_2,st)\;(\hat{s}_1,\hat{s}_2',\hat{st})$$

$$\mathsf{imove}_{1,2}(\Delta)\;v\,(v_1,v_2)(s_1,s_2,st)\;(\hat{s}_1,\hat{s}_2,\hat{st}) \;\doteq\; \text{let } \hat{st}' = \mathsf{newStat}(\hat{st},\hat{s}_1,\hat{s}_2) \text{ in}$$
$$\exists \hat{s}_1',\hat{s}_2'\in\mathsf{State}_{\mathsf{van}}.\;\hat{s}_1\Rightarrow\hat{s}_1'\;\wedge\;\hat{s}_2\Rightarrow\hat{s}_2'\;\wedge\;\Delta\;v\,(v_1,v_2)(s_1,s_2,st)\;(\hat{s}_1',\hat{s}_2',\hat{st}')$$

Fig. 7: Definition of $\mathsf{proact}(\Delta)$ – the proactive component of unwinding

$\mathsf{match}^{1,2}(\Delta)$. By contrast, subscripts refer to *angelic nondeterminism*: we are free to choose between different ways to react: either ignore, or match with one counterpart, or match with both counterparts; and indeed, e.g., $\mathsf{match}^{1,2}(\Delta)$ is defined as a *disjunction* (with side conditions) involving $\mathsf{match}^{1,2}_1(\Delta)$, $\mathsf{match}^{1,2}_2(\Delta)$ and $\mathsf{match}^{1,2}_{1,2}(\Delta)$.

Reactive growth. Predicate $\mathsf{react}(\Delta)$ (see Fig. 6) describes how the vstates $\hat{s}_1$ and $\hat{s}_2$ can transit by matching transitions of the ostates s_1 and s_2:

- either separately, via predicates $\mathsf{match}^1(\Delta)$ or $\mathsf{match}^2(\Delta)$, in case one of the ostates transits without any interaction (i.e., $\neg\ \mathsf{isInt}(s_1)$ or $\neg\ \mathsf{isInt}(s_2)$),
- or synchronously, via $\mathsf{match}^{1,2}(\Delta)$, in case both ostates transit interactively (i.e., $\mathsf{isInt}(s_1)$ and $\mathsf{isInt}(s_2)$).

There is ample flexibility when choosing the matching transitions. For example, let us detail the case of $\mathsf{match}^1(\Delta)$; those of $\mathsf{match}^2(\Delta)$ and $\mathsf{match}^{1,2}(\Delta)$ are similar. In the definition of $\mathsf{match}^1(\Delta)$, we are free to match a transition $s_1 \Rightarrow s'_1$ in one of three ways, as reflected by the disjunct on the righthand side of the implication from $s_1 \Rightarrow s'_1$:

1) either ignoring the transition, but this only provided no secret is produced by the target ($\neg\ \mathsf{isSec}(s_1)$)—otherwise the vtrace would have been forced to also produce a secret to avoid breaching the secrecy contract;
2) or giving a matching transition by the counterpart vtrace, $\hat{s}_1 \Rightarrow \hat{s}'_1$, via $\mathsf{match}^1_1(\Delta)$, provided the same secret (if any) is produced (i.e., $\mathsf{eqSec}(s_1, \hat{s}_1)$), and no vtrace interaction happens (i.e., $\neg\ \mathsf{isInt}(\hat{s}_1)$)—to ensure that the vtrace does not breach the interaction contract;
3) or giving matching transitions by both vtraces, $\hat{s}_1 \Rightarrow \hat{s}'_1$ and $\hat{s}_2 \Rightarrow \hat{s}'_2$, via $\mathsf{match}^1_{1,2}(\Delta)$, provided $\hat{s}_1$ respects the secrecy contract towards its counterpart (i.e., $\mathsf{eqSec}(s_1, \hat{s}_1)$), $\hat{s}_1$ and $\hat{s}_2$ respect the interaction contract towards each other (i.e., $\mathsf{eqAct}(\hat{s}_1, \hat{s}_2)$), and $\hat{s}_2$ doesn't produce a secret (not to breach the secrecy contract).

The predicate $\mathsf{match}^{1,2}$ is different from match^1 and match^2 in that it allows interaction ($\mathsf{isInt}(s_1)$ and $\mathsf{isInt}(s_2)$), which can lead to a change of the observation divergence status ($st' = \mathsf{newStat}(st, s_1, s_2)$), meaning that the otraces might *right now* diverge observationally. In this case, relative security requires that the vtraces also produce different observations, which is reflected in our unwinding condition. Indeed, according to the definition of $\mathsf{match}^{1,2}(\Delta)$, if the status has not changed, *or* divergence had already been recorded before in the vtraces (i.e., $st' = st \vee \hat{st} = \mathsf{diff}$), then one is allowed to react by either ignoring the transition or matching it with only one of the vtraces. However, if the status has changed then one must react with both vtraces transitioning, i.e., take the fourth, $\mathsf{match}^{1,2}_{1,2}(\Delta)$ option, and make sure that that vtrace divergence status has become diff in case the otraces status has ($st' = \mathsf{diff} \longrightarrow \hat{st}' = \mathsf{diff}$)—as seen in the definition of $\mathsf{match}^{1,2}_{1,2}(\Delta)$.

Proactive growth. Predicate $\mathsf{proact}(\Delta)$ (see Fig. 7) allows the vtraces to take transitions independently. Thus, proactive moves represent "extra help" when proving relative security. They have similar conditions as the reactive moves, but simpler since they involve only the vstates, not the ostates. Proactive, "independent" moves can be taken either by one vstate ($\mathsf{imove}_1(\Delta)$ or $\mathsf{imove}_2(\Delta)$), or by both ($\mathsf{imove}_{1,2}(\Delta)$), subject to restrictions of not producing secrets (to avoid breaching the secrecy contract) and producing the same action if any (or not at all if moving separately). In the case of mutual independent moves, $\mathsf{imove}_{1,2}(\Delta)$, a possible change of observation status can occur—which is being recorded because, in a presumptive proof of unwinding, it is helpful to know if the vtrace observations have already diverged, so that in the future the proof no longer has to be "on watch" for the otraces to diverge (so to have the vtraces diverge at the same time, as discussed above for $\mathsf{match}^{1,2}_{1,2}(\Delta)$).

Another aspect that distinguishes proactive moves from reactive moves is that the former do not advance the otraces, which means that they should not be allowed to proceed indefinitely thus "filibustering" the unwinding game. Indeed, in that case, unwinding would fail to ensure relative security, because the otraces may end up not being entirely processed, rendering the secrecy-contract conditions $\mathsf{S}(\hat{\pi}_i) = \mathsf{S}(\pi_i)$ uncertain. For this reason, we carry an additional timer parameter $v \in \mathbb{N}_\infty$ that **1)** is forced to decrease each time we take a proactive move (as seen in Def. 3 when passing to $\mathsf{proact}(\Delta)$ a value $w < v$) and **2)** is reset to ∞ when we take a reactive move, and stays ∞ during reactive moves. This ensures that the proactive moves cannot be taken continuously and infinitely, but eventually yield to reactive moves.

Upgrading to general unwinding. In the presence of infinite traces, there are more opportunities for "filibustering" in addition to the above discussed one coming from unbounded proactive moves (by the vtraces), namely from the unbalanced reactive moves where one of the vtraces $\hat{\pi}_i$ grows indefinitely and "starves" the other one. This would allow us to successfully play the unwinding game without proving relative security, more precisely without proving that the constructed $\hat{\pi}_1$ and $\hat{\pi}_2$ have the same observations. To counter this, we must ensure that either vtrace always eventually yields to the other one when taking reactive moves (unless that other one has terminated). This is the purpose of the additional timers v_1 and v_2 highlighted in Def. 3 and Figs. 6 and 7. Namely, we think of v_i as counting the time until $\hat{\pi}_i$ will make progress in a reactive move, via matching—indeed, in Fig. 6 we see how v_i is decreased each time a matching action is taken without $\hat{s}_i$ taking a transition, and is reset to ∞ as soon as $\hat{s}_i$ takes a transition.

5.3. Soundness of the unwinding proof method

Unwinding seeks to provide sufficient local (state-based) conditions ensuring relative security. This is indeed the case:

Thm. 5 (The Proof Unwinding Theorem) Assume that:

- Δ is a (finitary) unwinding relation.
- $\Delta\ \infty\ (\infty, \infty)$ (resp. $\Delta\ \infty$) covers the initial states, i.e.: for all $s_1, s_2 \in \mathsf{State_{opt}}$ such that $\mathsf{istate}(s_1)$ and $\mathsf{istate}(s_2)$, there exist $\hat{s}_1, \hat{s}_2 \in \mathsf{State}$ such that $\mathsf{istate}(\hat{s}_1)$ and $\Delta\ \infty\ (\infty, \infty)\ (s_1, s_2, \mathsf{eq})\ (\hat{s}_1, \hat{s}_2, \mathsf{eq})$ (resp. $\Delta\ \infty\ (s_1, s_2, \mathsf{eq})\ (\hat{s}_1, \hat{s}_2, \mathsf{eq})$).

Then (finitary) relative security holds, i.e., $(\mathcal{SM}_{\mathsf{opt}}, \mathcal{AM}_{\mathsf{opt}}) \geq_{\checkmark} (\mathcal{SM}_{\mathsf{van}}, \mathcal{AM}_{\mathsf{van}})$ (resp. $(\mathcal{SM}_{\mathsf{opt}}, \mathcal{AM}_{\mathsf{opt}}) \geq^{\mathsf{fin}}_{\checkmark} (\mathcal{SM}_{\mathsf{van}}, \mathcal{AM}_{\mathsf{van}})$).

The sound proof method enabled by Thm. 5 is the following: To prove that relative security holds, it suffices to provide an unwinding relation that covers the initial states.

Whereas relative security and finitary relative security are incomparable (neither implies the other), general (non-finitary) unwinding implies finitary unwinding (by ignoring the v_i timers). So the general unwinding proof method ensures both relative security and finitary relative security.

5.4. Compositional, distributed unwinding

In practice, one might prefer to not define a single unwinding relation, i.e., a relation Δ shown to "unwind into itself", but a network of relations that unwind into each other:

Def. 6 An *unwinding network* is a tuple $(n, \mathsf{next}, \mathsf{Init}, \overline{\Delta})$ where $n \in \mathbb{N}$, $\mathsf{Init} \subseteq \{0,\ldots,n\}$ $\mathsf{next} : \{0,\ldots,n\} \to \mathcal{P}(\{0,\ldots,n\})$ and $\overline{\Delta} = (\Delta_i)_{i\in\{0,\ldots,n\}}$ are such that, for all $i \in \{0,\ldots,n\}$ and $v, v_1, v_2, s_1, s_2, st, \hat{s}_1, \hat{s}_2, \hat{st}$, if $\Delta_i\ v\ (v_1, v_2)\ (s_1, s_2, st)\ (\hat{s}_1, \hat{s}_2, \hat{st})$ holds, then the conditions from Def. 3 hold but for Δ replaced with $\bigvee_{j\in\mathsf{next}(i)} \Delta_j$.

This definition generalizes Def. 3 by allowing each Δ_i to take reactive and proactive steps unwinding into any Δ_j to which Δ_i is connected, as described by the next operator. Init stands for set of the initial nodes in this network.

Thm. 7 (The Proof Distributed-Unwinding Theorem) Assume that $(n, \mathsf{next}, \overline{\Delta}, M)$ is an unwinding network and $\bigcup_{i\in\mathsf{Init}}(\Delta_i\ \infty)$ covers the initial states (as in Thm. 5). Then $(\mathcal{SM}_{\mathsf{opt}}, \mathcal{AM}_{\mathsf{opt}}) \geq_{\checkmark} (\mathcal{SM}_{\mathsf{van}}, \mathcal{AM}_{\mathsf{van}})$.

(And a similar generalization holds for finitary unwinding.)

Thm. 7 was useful in our verification case studies (to be described in §6), where different unwinding components turned out to naturally correspond to the different phases that the considered programs' executions go through.

5.5. Disproof Method for Relative Security

A counterexample for relative security requires **1)** a concrete step, providing a pair (π_1, π_2) of otraces that have the same actions and different observations, and **2)** an abstract reasoning step, showing that there is no similarly related pair $(\hat{\pi}_1, \hat{\pi}_2)$ of vtraces producing the same secrets as (π_1, π_2). For handling step 2, we can again use an unwinding-like technique. After collecting the sequences of secrets (σ_1, σ_2) of (π_1, π_2), we make sure that no suitable vtraces $(\hat{\pi}_1, \hat{\pi}_2)$ can cover these secrets. This is done by maintaining a four-place unwinding relation containing pairs (s_1, σ_1) and (s_2, σ_2), where each s_i is the state currently reached by the presumptive trace $\hat{\pi}_i$ and σ_i is the sequence of secrets still left to be covered by $\hat{\pi}_i$. We will state unwinding conditions ensuring that no vtraces starting in s_1 and s_2 can achieve both same actions and different observations if they are to stay on track, i.e., cover the given secrets σ_1 and σ_2—in this sense, our unwinding relations will be secret-directed.

Def. 8 A relation $\Gamma : (\mathsf{State}_{\mathsf{van}} \times \mathsf{Seq}(\mathsf{Sec})) \to (\mathsf{State}_{\mathsf{van}} \times \mathsf{Seq}(\mathsf{Sec})) \to \mathsf{Bool}$ is a *secret-directed (SD) unwinding* when for all $s_1, \sigma_1, s_2, \sigma_2$, if $\Gamma\ (s_1, \sigma_1)\ (s_2, \sigma_2)$ then:

- $\mathsf{isInt}(s_1) \leftrightarrow \mathsf{isInt}(s_2)$
- $\neg\ \mathsf{isInt}(s_1) \longrightarrow$ $\mathsf{move}_1(\Gamma)\ (s_1, \sigma_1)\ (s_2, \sigma_2) \wedge \mathsf{move}_2(\Gamma)\ (s_1, \sigma_1)\ (s_2, \sigma_2)$
- $\mathsf{isInt}(s_1) \wedge \mathsf{getAct}(s_1) = \mathsf{getAct}(s_2) \longrightarrow$ $\mathsf{getObs}(s_1) = \mathsf{getObs}(s_2) \wedge \mathsf{move}_{1,2}(\Gamma)\ (s_1, \sigma_1)\ (s_2, \sigma_2)$

The predicates $\mathsf{move}_1(\Gamma)$, $\mathsf{move}_2(\Gamma)$ and $\mathsf{move}_{1,2}(\Gamma)$ are defined in Fig. 8. They are based on the secret-directed transition relation $\Rightarrow^{\mathsf{getSec}}_{\mathsf{isSec}}$, also defined in Fig. 8.

As seen in the above bullet points, an SD unwinding guarantees that the two states s_1 and s_2 (reached by the presumptive vtraces) always have the same interaction status, and equal observations if equal actions; also, the secret-directed transition relation $\Rightarrow^{\mathsf{getSec}}_{\mathsf{isSec}}$ shown in Fig. 8 makes sure that the states s_1 and s_2 can evolve (i.e., the vtraces can grow) only by respecting the remaining sequences of secrets—i.e., only producing the next secret in the corresponding sequence, if at all producing a secret. And indeed, maintaining an SD unwinding while starting with the sequences of secrets $(\mathsf{S}(\pi_1), \mathsf{S}(\pi_2))$ given by two concrete otraces (π_1, π_2), constitutes a sound disproof method:

Thm. 9 (The Disproof Unwinding Theorem) Assuming:

- $\pi_1, \pi_2 \in \mathsf{Trace}_{\mathsf{opt}}$, $\mathsf{A}(\pi_1) = \mathsf{A}(\pi_2)$ and $\mathsf{O}(\pi_1) \neq \mathsf{O}(\pi_2)$.
- Γ is an SD unwinding.
- Γ covers the initial states w.r.t. $(\mathsf{S}(\pi_1), \mathsf{S}(\pi_2))$, in that $\Gamma\ (s_1, \mathsf{S}(\pi_1))\ (s_2, \mathsf{S}(\pi_2))$ holds for all $s_1, s_2 \in \mathsf{State}_{\mathsf{van}}$ such that $\mathsf{istate}(s_1)$ and $\mathsf{istate}(s_2)$.

Then relative security fails, i.e., $(\mathcal{SM}_{\mathsf{opt}}, \mathcal{AM}_{\mathsf{opt}}) \not\geq_{\checkmark} (\mathcal{AM}_{\mathsf{van}}, \mathcal{LM}_{\mathsf{van}})$. And if $\pi_1, \pi_2 \in \mathsf{Trace}^{\mathsf{fin}}$, then finitary relative security also fails, i.e., $(\mathcal{SM}_{\mathsf{opt}}, \mathcal{AM}_{\mathsf{opt}}) \not\geq^{\mathsf{fin}}_{\checkmark} (\mathcal{SM}_{\mathsf{van}}, \mathcal{AM}_{\mathsf{van}})$.

6. Unwinding (Dis)Proofs for our Examples

Unwinding proofs. We deployed the distributed unwinding method (Thm. 7) to prove the relative security of (§3.2's IMP representations of) `fun2`–`fun6`. The proofs turn the informal intuition for why these programs are (relatively) secure into an unwinding argument involving ostates s_1 and s_2 (those reached by two otraces π_1 and π_2) and vstates $\hat{s}_1$ and $\hat{s}_2$ (those reached by two vtraces $\hat{\pi}_1$ and $\hat{\pi}_2$).

In all the proofs, the two ostates s_1 and s_2 always have the same PC (i.e., execute the statements lock-step synchronously) at all levels; and the two vstates $\hat{s}_1$ and $\hat{s}_2$ always have the same PC too. Moreover, often (but not always) the vstates have the same PC with the level-0 PC of the ostates. Since interaction is pervasive, the match^1 and match^2 predicates are vacuously true, and when checking $\mathsf{match}^{1,2}$ we only choose "ignore" (first disjunct) or the $\mathsf{match}^{1,2}_{1,2}$ (fourth disjunct) options—this is because of the aforementioned lock-step synchronization. Another common aspect will be that s_1 and s_2 always have the same value for the input variable x, and we also make sure that $\hat{s}_1$ and $\hat{s}_2$ have the same value for x—this is to respect

$$(s,\sigma) \Rightarrow^{\mathsf{getSec}}_{\mathsf{isSec}} (s',\sigma') \doteq s \Rightarrow s' \wedge ((\neg\mathsf{isSec}(s) \wedge \sigma = \sigma') \vee (\mathsf{isSec}(s) \wedge \sigma = \mathsf{getSec}(s)\cdot\sigma'))$$
$$\mathsf{move}_1(\Gamma)\ (s_1,\sigma_1)\ (s_2,\sigma_2) \doteq \forall s_1',\sigma_1'.\ (s_1,\sigma_1) \Rightarrow^{\mathsf{getSec}}_{\mathsf{isSec}} (s_1',\sigma_1') \longrightarrow \Gamma\ (s_1',\sigma_1')\ (s_2,\sigma_2)$$
$$\mathsf{move}_2(\Gamma)\ (s_1,\sigma_1)\ (s_2,\sigma_2) \doteq \forall s_2',\sigma_2'.\ (s_2,\sigma_2) \Rightarrow^{\mathsf{getSec}}_{\mathsf{isSec}} (s_2',\sigma_2') \longrightarrow \Gamma\ (s_1,\sigma_1)\ (s_2',\sigma_2')$$
$$\mathsf{move}_{1,2}(\Gamma)\ (s_1,\sigma_1)\ (s_2,\sigma_2) \doteq \forall s_1',\sigma_1',s_2',\sigma_2'.\ (s_1,\sigma_1) \Rightarrow^{\mathsf{getSec}}_{\mathsf{isSec}} (s_1',\sigma_1') \wedge (s_2,\sigma_2) \Rightarrow^{\mathsf{getSec}}_{\mathsf{isSec}} (s_2',\sigma_2') \longrightarrow \Gamma\ (s_1',\sigma_1')\ (s_2',\sigma_2')$$

Fig. 8: The defining predicates for SD unwinding

the interaction contract, namely the "action" part of this contract, which assumes that $\mathsf{A}(\pi_1) = \mathsf{A}(\pi_2)$ and guarantees $\mathsf{A}(\hat{\pi}_1) = \mathsf{A}(\hat{\pi}_2)$. Often (but not always) this value of x is the same across the board, i.e., the same for $\hat{s}_i$ as for s_i.

Note that s_1 and s_2 are allowed to differ in the initial secrets, i.e., in the values stored in the arrays a and b. According to the secrecy contract, we always keep $\hat{s}_1$ have the same values in a and b as s_1, and similarly for $\hat{s}_2$ versus s_2. Our unwinding relations (unwindings for short) need to ensure that, whenever an output statement gets executed, if the outputted value or the set of read locations differs for s_1 versus s_2 then it must also differ for $\hat{s}_1$ versus $\hat{s}_2$—this is here the observation part of the interaction contract.

`fun2` is secure essentially because the early fence on line 4 immediately inhibits any (mis)speculation that goes on the "then" branch of the "if" conditional. This is reflected by our unwindings keeping all PCs synchronized and all values of x the same, which means that the interaction contract is maintained straightforwardly.

`fun3`'s proof is slightly more subtle: In the only interesting scenario, speculation incorrectly takes the "then" branch, and now the otraces access locations that the vtraces cannot access, namely the x'th location of a. But since s_1 and s_2 have the same value for x, the otraces access the same locations (among each other), so they are not observationally "more different" than the vtraces are—and this is the invariant we keep in our unwindings, which again validates the interaction contract.

For `fun4`, assuming $N > 0$, we must use two unwinding strategies depending on whether the value of the input is $< N$ or not. The first case is not problematic, because there otraces can only diverge from vtraces in an immediately harmless way, i.e., by (mis)speculating on the "else" branch. For the second case, of $x \geq N$, (mis)speculation can go on the "then" branch and cause observational "damage" to the otraces, which access different locations from b depending on the value of $a[0]$, on which they may differ. To counter this, we choose a different input for the vtraces, namely 0. This means that the vtraces also take the "then" branch, keeping thier PCs synchronized with the level-1 PCs of the otraces. At some point, speculation will be abandoned and the otraces will go back to the correct branch, reaching the output statement (i.e., s_1 and s_2 will be back at speculation level 0 and PC the one of the output statement). This is the moment when, in our unwinding, $\hat{s}_1$ and $\hat{s}_2$ will take one or two proactive actions (depending on where the PCs were on the "then" branch when the speculation was abandoned by s_1 and s_2), using the proact option of the unwinding with the $\mathsf{imove}^{1,2}$ option to also finish the "then" branch and reach the output statement. After this, the ostates are again PC-synchronized with the vstates, and the output statement can be taken in lockstep—while, importantly, $\hat{s}_1$ and $\hat{s}_2$ differ observationally at least as much as s_1 and s_2 do.

Implicit above is that we use networks of unwindings (as in Thm. 7) instead of single unwindings. We have one unwinding component for each phase in the program execution (e.g., "before entering the if branch", or "at the fence") and we transit between components upon relevant events (e.g., "the if branch is taken", or "speculation starts").

Since `fun5` is essentially the while-iteration of `fun3` (where input statements are replaced by scanf and return statements by printf), its proof is obtained from that of `fun5` by adding corresponding cycles in the unwinding network that proves `fun4`, and additionally handling the speculation on the outer loop. This shows the potential of unwinding proofs following compositionally the syntactic structure.

Finally, the difference between `fun6` and `fun5` is that, in addition to the untrusted input x, `fun6` also takes a trusted input y which is considered to be secret. Our unwinding proof for `fun5` can be adapted to `fun6`, noticing that the required secrecy contract (enforced via the unwinding's eqSec predicate) still holds because, in the unwinding strategy, the $\mathsf{Input}_{\mathsf{T}}\, y$ statement is always executed synchronously by the otraces and vtraces.

A note on unwinding timer parameters: The "v" timer (which bounds proactive moves) stays in the range $\{\infty, 2, 1, 0\}$ since we never need to take more than 3 proactive moves in a row (and this is only when vtraces must "catch-up" with mispredicting otraces by finishing their branch). As for the "v_1, v_2" timers (which bound idle reactive moves), the same range suffices, since the only time this is needed is when mispredicting moves are ignored.

SD unwinding disproof. The intuitive reason why `fun1` is insecure is that, for $x \geq N$, speculation can access the $(a[x] * 512)$'th location of b whereas normal execution cannot. With our disproof method, this argument is formalized by choosing a value $i \geq N$ (as input for x) and two initial arrays a that differ on $a[i]$ but nowhere else. These give two (singleton) sequences of secrets (the $\mathsf{S}(\pi_1)$ and $\mathsf{S}(\pi_2)$ in Thm. 9), and two otraces π_1 and π_2 that produce these secrets (i.e., start in memories with the respective arrays). Then we show that any two vtraces that produce these secrets (i.e., start in the two given memories) must be observationally equal (i.e., output the same value) provided they are action-equal (i.e., input the same value x). Indeed, here a trivial secret-directed unwinding keeps the two potential vtraces completely synchronized; they can only take the "then" branch if x is smaller than N, which means that x is different from i, which further means that what the two vtraces print cannot depend on $a[i]$ hence must be equal.

7. Isabelle Mechanization

We used the Isabelle/HOL theorem prover [25] to mechanize the relative security framework and the examples [7].

Theory structure and sizes. The theory structure of the entire mechanization is shown in Fig. 9. The theories formalizing the abstract framework are shown in the middle-right part of the figure, e.g., Relative_Security, Unwinding etc. The suffix "fin" refers to the finitary (finite-trace) version of the concepts. These theories consist of 11K LOC, with the largest part (7.5K LOC) taken by infinitary unwinding—which also required some substantial preparations (part of Trivia, 2.6K LOC) dedicated to extending the coinductive (lazy) list library with custom support for the inductive-coinductive mixture of reasoning called for by the timers.

The theories Syntax, Step_Basic, Step_Normal and Step_Spec, taking 1.5K LOC, formalize IMP's syntax and semantics straightforwardly, via datatypes and inductive predicates. The instantiation of relative security to IMP, taking 0.6K LOC, is formalized in theory Instance_Common, covering the interaction infrastructure (which is common to all examples), and theories Instance_Secret_IMem and Instance_Secret_IMem_Inp, covering the two secrecy infrastructures: one with secrets as the initial memories (for fun1–fun5), and one with additional secrets as trusted inputs and outputs (for fun6). We used finitary unwinding and SD-unwinding for verifying the programs that are known to be terminating, and used the heavier infinitary unwinding only for the possibly nonterminating fun5 and fun6. Verifying the (in)security of fun1–fun6 took an average of 2K LOC per example, which splits into 0.7K LOC for simplification-rules boilerplate (theories named fun⟨i⟩) and 1.3 LOC for the distributed unwinding proof (theories named fun⟨i⟩(in)secure). These large proof sizes, clearly unfeasible in practice, are due partly to the boilerplate needed to make up for Isabelle not being a program logic, and partly to the heaviness of unwinding itself. In the future, this could be alleviated via custom tactics for automating the boilerplate.

Locale structure. Our mechanization relies heavily on Isabelle's locales [16]. A locale fixes types, constants and assumptions, allowing one to perform definitions and proofs relative to these. A locale L can be *interpreted* at the top level of an Isabelle theory by instantiating its types and constants, and verifying its assumptions. It can also be interpreted relatively to another (more concrete) locale L' by establishing a *sublocale* relationship $L' \leq L$.

We used (sub)locales to formalize our gradual move from abstract to concrete. We have a locale System_Mod for system models, extended by a locale Leakage_Mod that adds a type Leak and a predicate leakVia. Moreover, we have the locales Statewise_Attacker_Mod and Attacker_Mod forming an abstract-towards-concrete sublocale hierarchy Leakage_Mod < Attacker_Mod < Statewise_Attacker_Mod.

Relative security is first defined abstractly by importing two (disjoint) copies of Leakage_Mod (labeled "Van" and "Opt") and then made more concrete along the above hierarchy, leading to the locale Rel_Sec which imports two copies of Statewise_Attacker_Mod. It is in Rel_Sec where §5's unwinding (dis)proof methods are proved sound, following the ideas described in §5.1, §5.2, §5.5 when motivating (the soundness of) our unwinding conditions.

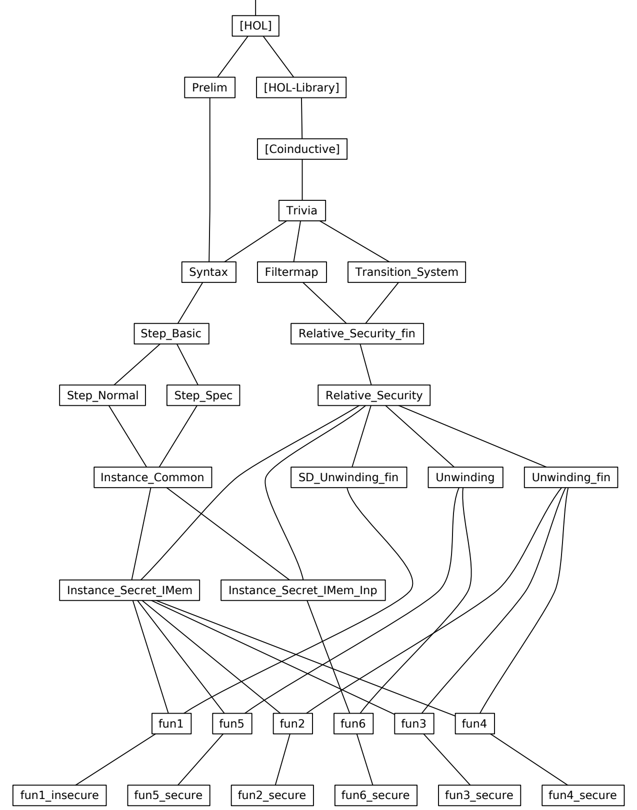

Fig. 9: Theory structure of our Isabelle mechanization

The instantiation of the locale Rel_Sec, to IMP programs was performed by a sublocale relationship Prog_Mispred_Init < Rel_Sec, where Prog_Mispred_Init is a locale that fixes a misprediction oracle (like in §3.1), a program and some initial memory, inputs and predictor state (like in §3.3). Finally, Prog_Mispred_Init is instantiated (via locale interpretation) to the fun1–fun6 examples; this makes available the unwinding (dis)proof methods from Rel_Sec (via Prog_Mispred_Init), and the relative (in)security of these examples is proved along the lines sketched in §6.

8. Further Related Work

Our framework's innovation compared to previous work is in regarding leaks as first-class citizens and allowing fully interactive attackers and dynamic secrets (see §4). Proof-theoretically, the framework generalizes the unwinding method [11], which was widely deployed in the security literature [20, 21, 24, 26, 27]. Our work is the first to develop an unwinding method not for proving the (absolute) security of a system, but for comparing the security of two systems (necessarily a four-trace property), and to propose unwinding for disproofs as well. This unwinding-based foundation

could underpin automatic and compositional analysis/verification methods, such as type systems [2, 29, 31, 32].

Due to the need to consider alternative execution traces, proving information-flow security properties has similarities with proving program equivalence, where techniques involving symbolic execution and bisimulation have been proposed [1, 9, 15]. Relative security seems related to Morgan's refinement order for noninterference [22, 23]; working out the formal connection between the two notions could lead to more insights into compositional proofs for our notion.

Acknowledgments

We thank the reviewers for their excellent suggestions for improvement, and for catching several technical typos. We gratefully acknowledge support from the EPSRC grants EP/X015114/1 and EP/X015149/1, titled "Safe and secure COncurrent programming for adVancEd aRchiTectures (COVERT)".

References

[1] G. Barthe, J. M. Crespo, and C. Kunz, "Relational verification using product programs," in *FM*, ser. LNCS, vol. 6664, 2011, pp. 200–214.
[2] G. Boudol and I. Castellani, "Noninterference for concurrent programs and thread systems," *Theor. Comp. Sci.*, vol. 281, no. 1-2, pp. 109–130, 2002.
[3] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *USENIX Sec.*, 2019, pp. 249–266.
[4] S. Cauligi, C. Disselkoen, D. Moghimi, G. Barthe, and D. Stefan, "Sok: Practical foundations for software spectre defenses," in *SP*. IEEE, 2022, pp. 666–680.
[5] K. Cheang, C. Rasmussen, S. A. Seshia, and P. Subramanyan, "A formal approach to secure speculation," in *CSF*. IEEE, 2019, pp. 288–303.
[6] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *J. Com. Sec.*, vol. 18, no. 6, pp. 1157–1210, 2010.
[7] B. Dongol, M. Griffin, A. Popescu, and J. Wright, "Isabelle mechanization of relative security," https://www.andreipopescu.uk/RelSec.zip.
[8] B. Dongol, M. Griffin, A. Popescu, and J. Wright, "Relative security: Formally modeling and (dis)proving resilience against semantic optimization vulnerabilities," Extended technical report. TODO: URL.
[9] B. Godlin and O. Strichman, "Regression verification: proving the equivalence of similar programs," *Softw. Test. Verif. Reliab.*, vol. 23, no. 3, pp. 241–258, 2013.
[10] J. A. Goguen and J. Meseguer, "Security policies and security models," in *SP*, 1982, pp. 11–20.
[11] J. A. Goguen and J. Meseguer, "Unwinding and inference control," in *SP*, 1984, pp. 75–87.
[12] R. Guanciale, M. Balliu, and M. Dam, "Inspectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis," in *CCS*, 2020, pp. 1853–1869.
[13] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "Spectector: Principled detection of speculative information flows," in *SP*. IEEE, 2020, pp. 1–19.
[14] M. Guarnieri, B. Köpf, J. Reineke, and P. Vila, "Hardware-software contracts for secure speculation," in *SP*. IEEE, 2021, pp. 1868–1883.
[15] C. Hur, G. Neis, D. Dreyer, and V. Vafeiadis, "The power of parameterization in coinductive proof," in *POPL*. ACM, 2013, pp. 193–206.
[16] F. Kammüller, M. Wenzel, and L. C. Paulson, "Locales—a sectioning concept for Isabelle," in *TPHOLs*, 1999, pp. 149–166.
[17] P. Kocher, "Spectre mitigations in Microsoft's C/C++ compiler," 2018, https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html.
[18] 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 *SP*. IEEE, 2019, pp. 1–19.
[19] 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 Sec.*, 2018.
[20] H. Mantel, "A uniform framework for the formal specification and verification of information flow security," Ph.D. dissertation, University of Saarbrücken, 2003.
[21] J. McLean, "A general theory of composition for trace sets closed under selective interleaving functions," in *SP*, 1994, pp. 79–93.
[22] C. Morgan, "The shadow knows: Refinement and security in sequential programs," *Sci. Comput. Program.*, vol. 74, no. 8, pp. 629–653, 2009.
[23] C. Morgan, "Compositional noninterference from first principles," *Form. Asp. Comput.*, vol. 24, no. 1, pp. 3–26, 2012.
[24] T. C. Murray, D. Matichuk, M. Brassil, P. Gammie, and G. Klein, "Noninterference for operating system kernels," in *CPP*, 2012, pp. 126–142.
[25] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002.
[26] A. Popescu, P. Lammich, and T. Bauereiss, "Cocon: A confidentiality-verified conference management system," *Arch. Formal Proofs*, 2021.
[27] J. Rushby, "Noninterference, transitivity, and channel-control security policies," SRI, Tech. Rep., Dec 1992, http://www.csl.sri.com/papers/csl-92-2/.
[28] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Sel. Areas in Comm.*, vol. 21, no. 1, pp. 5–19, 2003.
[29] A. Sabelfeld and D. Sands, "Probabilistic noninterference for multi-threaded programs," in *IEEE Computer Security Foundations Workshop*, 2000, pp. 200–214.
[30] D. Sangiorgi, "On the bisimulation proof method," *Math. Struc. Com. Sci.*, vol. 8, no. 5, pp. 447–479, 1998.

[31] B. A. Shivakumar, G. Barthe, B. Grégoire, V. Laporte, T. Oliveira, S. Priya, P. Schwabe, and L. Tabary-Maujean, “Typing high-speed cryptography against spectre v1,” in *SP*. IEEE, 2023, pp. 1094–1111.
[32] D. Volpano and G. Smith, “Probabilistic noninterference in a concurrent language,” *Journal of Computer Security*, vol. 7, no. 2,3, pp. 231–253, 1999.
[33] S. Zdancewic and A. C. Myers, “Observational determinism for concurrent program security,” in *IEEE CSF Workshop*, 2003, pp. 29–43.