Journal of Automated Reasoning (2025) 69:31
https://doi.org/10.1007/s10817-025-09744-7

Relative Security: (Dis)Proving Resilience Against Semantic Optimization Vulnerabilities in Isabelle/HOL

Extended Version

John Derrick[1] · Brijesh Dongol[2] · Chelsea Edmonds[1] · Matt Griffin[2,3] · Andrei Popescu[1] · Jamie Wright[1]

Received: 17 February 2025 / Accepted: 24 October 2025 / Published online: 18 November 2025

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. This paper is an extension of an earlier conference paper that provides significantly more detail on the Isabelle formalization and the unwinding proof process.

Keywords Relative security · Unwinding · Isabelle/HOL · Information-flow security

The authors are listed alphabetically, regardless of contribution or seniority.

✉ John Derrick
j.derrick@sheffield.ac.uk

✉ Brijesh Dongol
b.dongol@surrey.ac.uk

✉ Chelsea Edmonds
c.l.edmonds@sheffield.ac.uk

✉ Matt Griffin
matthew.griffin@imperial.ac.uk

✉ Andrei Popescu
a.popescu@sheffield.ac.uk

✉ Jamie Wright
jwright8@sheffield.ac.uk

[1] Department of Computer Science, University of Sheffield, Sheffield, South Yorkshire, England

[2] Department of Computer Science, University of Surrey, Guildford, England

[3] Department of Computer Science, Imperial College London, London, UK

1 Introduction

Meltdown [1] and Spectre [2] 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.

Since their discovery, Meltdown, Spectre and their variants have sparked intensive research interest, leading to a range of mitigation techniques [3]. While Meltdown-based attacks have now largely been resolved using a combination of hardware- or microcode-based patches [3], older architectures remain vulnerable. Moreover, for Spectre and its variants, new generations of attacks are subtle and difficult to detect. This problem is of high concern since vulnerabilities can leak sensitive information across hypervisor domains, breaking virtual machine boundaries. Since attacks can be carried out using valid processor mechanisms and usually leave no discernable traces, their detection is extremely hard. Principled approaches to formally expressing and verifying the absence of such vulnerabilities are therefore of primary importance.

This paper proposes a new model- and proof-theoretic framework for describing and verifying resilience against transient execution attacks and beyond. We introduce *relative security*, 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* increase in information leakage: any leak occurring in the optimization-enhanced system can also occur in the vanilla one.

While building on a rich literature that addresses this problem from a formal modeling perspective, 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 a set of general methods for *proving* and *disproving* relative security in an incremental fashion, generalizing the unwinding method by Goguen and Meseguer [4]. On the proof front, 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 dissimilarities between pairs of corresponding traces.

Due to the complex dynamics between the four involved traces, the *disproof* front of relative security is also interesting and benefits from the idea of unwinding. A counterexample involves 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.

To demonstrate the applicability of both our definition of relative security and the associated unwinding (dis)proof methods, we instantiate relative security to a programming language with speculative semantics. This yields intuitive results on some standard example programs, which the proof methods are then applied to.

The relative security framework, as well as the language semantics and examples, have been mechanized in the Isabelle/HOL prover [5]. The formal libraries these extensions are based off are available in the Isabelle Archive of Formal Proofs [6–8].

This journal paper is a considerable extension on an earlier conference paper [9] with a greater focus on the formalization and its role in the proofs. The key differences are as follows.

1. Both of the sections that are focused on the formalization (§4 and §7) are new.
2. There are significant extensions to the descriptions of abstract unwinding (§5), including further details of the unwinding definitions, soundness proofs, and links with the formalization.
3. There are significant extensions to the discussion on each case study and their (dis)proofs of (in)correctness (§8).
4. There are additional discussions on related work from the perspective of formalization in a theorem prover (§9).

Overview The first half of this paper focuses on the abstract contributions, including the motivating examples (§2), the definition of relative security (§3) and its formalization (§4), as well as the development of the unwinding proof method including the abstract soundness proofs (§5). The latter half focuses on the concrete instantiations of these concepts based on a basic language that models speculative execution (§6) and its formalization (§7), which is then used to demonstrate the application of our unwinding (dis)proof methods to several case studies (§8). We conclude by presenting a discussion of related work (§9), and summary of our main contributions (§10).

2 Motivating Examples

This section motivates our notion of relative security by introducing example programs that exhibit various features that we would like to cover. We focus on C programs, including some standard examples from the Spectre literature [10, 11]. 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 the Spectre attacks).

Spectre v1 attack Our examples pertain to the Spectre v1 attack (aka *bounds check bypass*), which proceeds in three phases. We describe these phases in terms of the example in Listing 1. We assume that the attacker calls `fun1` with an input of its choosing.

Phase 1. The attacker (mis)trains the branch predictor by calling `fun1` with values less than `N`, making it more likely that it will enter the true branch of the `if` statement at line 3.

Phase 2. The attacker calls `fun1` with `x` $\geq$ `N`, aiming to trigger misspeculation and access the secret stored in `&a + x`, where `&a` is the address of array `a`. If misspeculation is triggered, the program fetches the (secret) value, say *sec*, from `&a + x`, and *sec* is subsequently used to access some other value of array `b`. (Here, the multiplication by `512` ensures that this value is stored in its own cache line, making the attack more efficient.) Note that the value of `b[`s `* 512]` is unimportant. What is important is that after resolving speculation, the cached content of `b` is not cleared, which means that it is possible to perform a timing attack (Phase 3).

Phase 3. The attacker performs a standard timing attack, polling `b[`i `* 512]` for different values of i. On $i = sec$, it can notice that the value returns much faster, allowing it to determine *sec*.

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

Listing 1 Spectre Bounds Check Bypass (BCB)

In this paper, we are interested in mitigation steps for Phase 2, whereby we prevent programs from performing "dangerous" misspeculation.

```
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

```
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

A suboptimal fix to `fun1` 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. This less obvious fix is more optimal than the one shown in `fun2` since the delayed fence allows for more speculation to occur. In general, we want to place fences as late as possible within branches in order to maximize speculation.

Conditionally secure example

Now consider the program in Listing 4, discussed by Cheang et al. [11]. 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.
- For `N > 0`, consider the following. Suppose `N = 2` (for concreteness) 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.

```
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

Interactive examples

The examples above 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 continually takes an integer value as input (via the `scanf` function) and accesses the arrays `a` and `b` based on the input until its value is 0. It also outputs a (possibly infinite) stream of elements from the array `b` (via the `printf` function). 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*. Despite this behavior, we wish to show that the program (e.g., Listing 5) is still Spectre-secure, namely that the fence in line 8 is sufficient to prevent transient execution vulnerabilities.

```
1  void fun5() {
2    unsigned t = 0;
3    unsigned x = 1;
4    while (x != 0) {
5      scanf("%u", &x);
6      if (x < N) {
7        unsigned v = a[x];
8        _mm_lfence();
9        t = b[v * 512];
10       printf("%u", t);
11     }
12   }
13 }
```

Listing 5 Secure interactive program

```
1  void fun6() {
2    unsigned t = 0;
3    unsigned x = 1;
4    while (x != 0) {
5      x = getUntrustedInput();
6      unsigned y =
       getTrustedInput();
7      if (x < N) {
8        unsigned v = a[x];
9        writeOnSecretFile(x,y);
10       _mm_lfence();
11       t = b[v * 512];
12       printf("%u", t);
13     }
14   }
15 }
```

Listing 6 Secure secret-interactive program

In some cases, one may wish to protect not only the initial memory, but also (secret) inputs that are under the control of trusted parties. Listing 6 shows a contrived example illustrating this: the program reads from both an untrusted and a trusted source. The call to a procedure named `writeOnSecretFile` illustrates the potential processing of some trusted input stored in `y`, 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, i.e., we can ensure that the input from the trusted source is not leaked.

In conclusion, we want to model fully interactive (both acting and observing) attackers and fully interactive secret uploading, while also allowing interaction to take place in(de)finitely, i.e., factoring in infinite executions.

3 Defining Relative Security

We develop the abstract framework on top of system models capturing the program semantics (§3.1). Our approach begins with the abstract notion of *leakage models* that allow us to express the essence of relative security (§3.2). We gradually refine this definition to more concrete *(state-wise) attacker models* (§3.3, §3.4), where leaks are defined from secrets together with attacker actions and observations, which in turn can later be easily instantiated by our concrete case studies (§8).

3.1 System Model

We first introduce the concept of a *system model*, which provides a base from which we can later model leaks and attackers:

Def 1 A *system model* is a triple $\mathcal{SM} = (\mathsf{State}, \mathsf{istate}, \Rightarrow)$, consisting of:

- a set State of *states*, ranged over by s,
- a predicate $\mathsf{istate} : \mathsf{State} \to \mathsf{Bool}$ that describes the *initial states*,
- a binary relation $\Rightarrow : \mathsf{State} \times \mathsf{State} \to \mathsf{Bool}$ on states called the *transition relation*.

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. Relative to a system model $\mathcal{SM} = (\mathsf{State}, \mathsf{istate}, \Rightarrow)$, we introduce the following notions:

- 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 non-empty (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.) $\mathsf{Trace} \subseteq \mathsf{Seq}(\mathsf{State})$, ranged over by π, ρ, denotes the set of traces, and $\mathsf{Trace}^{\mathsf{fin}}$ denotes the subset of Trace consisting of the finite traces only.

3.2 Very Abstract Relative Security: Leakage Models

We fix two system models: a *vanilla* system model, $\mathcal{SM}_{\mathsf{van}} = (\mathsf{State}_{\mathsf{van}}, \mathsf{istate}_{\mathsf{van}}, \Rightarrow_{\mathsf{van}})$, and an *optimization-enhanced* system model, $\mathcal{SM}_{\mathsf{opt}} = (\mathsf{State}_{\mathsf{opt}}, \mathsf{istate}_{\mathsf{opt}}, \Rightarrow_{\mathsf{opt}})$, such that $\mathsf{State}_{\mathsf{van}} \subseteq \mathsf{State}_{\mathsf{opt}}$. The vanilla system model represents the plain system, featuring no optimization—which will act as reference for our (relative) notion of security. Its set of traces is denoted by $\mathsf{Trace}_{\mathsf{van}}$. The optimization-enhanced system model stands 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 reason *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:

Def 2 A *leakage model* for a system model $\mathcal{SM} = (\mathsf{State}, \mathsf{istate}, \Rightarrow)$ is a pair $(\mathsf{Leak}, \mathsf{leakVia})$, where Leak, ranged over by l, is a set of entities called *leaks*, and $\mathsf{leakVia}$ 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 [12] that whatever the leaks are, one requires two traces (i.e., two alternative executions) rather than just one to exhibit it.

Note that we work very abstractly, gradually instantiating our concepts. For now, we leave the notion of a leak unspecified. In the next subsection (§3.3), 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 [13]. In (§6), when we have some concrete system models given by a programming language semantics at our disposal, 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 3 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:

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

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 serve as reminders 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., [14]) and pragmatic reasons (since, as we shall see, finitary security is amenable to a simpler unwinding proof method). $\geq^{\mathsf{fin}}_{\checkmark}$ is the same as $\geq_{\checkmark}$ for terminating programs Listings 1–4 in (§2), but $\geq_{\checkmark}$ is more suitable for possibly nonterminating interactive programs as in Listings 5 and 6.

3.3 Less Abstract Relative Security: Attacker Models

Relative security, as defined in §3.2, is parameterized by leaks. But still, what *is* a leak more concretely? To give a plausible answer, recall the key ingredients identified in our examples (§2): secret uploading and observer interaction, and our desire to capture fully interactive versions of these. This leads us to the next definition.

Def 4 An *attacker model* for a system model $\mathcal{SM} = (\mathsf{State}, \mathsf{istate}, \Rightarrow)$, 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, the attacker model indicates what needs protection (the secrets) and what attacker actions / observations are available. Together, the functions S, O and A naturally formalize the notion of leak. In particular, we instantiate $\mathsf{Leak} \doteq \mathsf{Seq}(\mathsf{Sec}) \times \mathsf{Seq}(\mathsf{Sec})$ and take the following instantiation of $\mathsf{leakVia}$:

$$\mathsf{leakVia}(\pi_1, \pi_2, (\sigma_1, \sigma_2)) \doteq \mathsf{S}(\pi_1) = \sigma_1 \ \wedge \ \mathsf{S}(\pi_2) = \sigma_2 \wedge \tag{1}$$

$$\mathsf{A}(\pi_1) = \mathsf{A}(\pi_2) \wedge \tag{2}$$

$$\mathsf{O}(\pi_1) \neq \mathsf{O}(\pi_2) \tag{3}$$

That is,

- by (1), π_1 and π_2 have the sequences of secrets σ_1 and σ_2,
- by (2), the attacker took the *same* actions during π_1 and π_2,
- which, by (3), led to the attacker making *different* observations.

In other words, this instantiation of $\mathsf{leakVia}(\pi_1, \pi_2, (\sigma_1, \sigma_2))$ 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 5 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 the 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:

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

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

The above just expands the aforementioned construction of leakage models from attacker models, so Def. 5 is a particular case of Def. 3. Its conclusion can be reformulated without explicitly quantifying over secrets as follows:

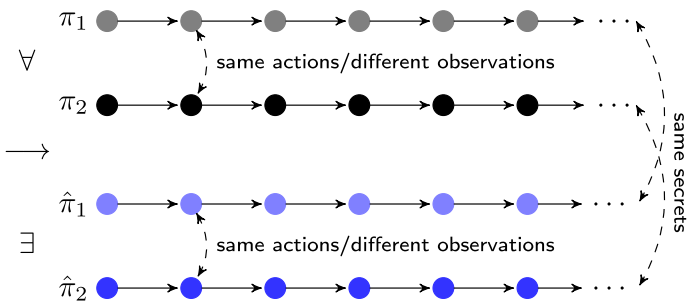


Fig. 1 Visualizing relative security.

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

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. For context, note that in our motivating examples (§2), 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)
- A will give any inputs on untrusted channels; and
- O will give any returned or printed values via untrusted channels, and the memory locations accessed for reading—see §6.3

Note that, while attacker models seem to be the most natural instantiation of the notion of leakage model, there are other potentially useful variations of this instantiation. For example, still taking leaks to be pairs of sequences of secrets (σ_1, σ_2), the $\mathsf{leakVia}$ predicate can be enriched so that $\mathsf{leakVia}(\pi_1, \pi_2, (\sigma_1, \sigma_2))$ adds further constraints to either the relationships between π_i and σ_i or to the relationship between σ_1 and σ_2—in the latter case allowing one to express declassification bounds similarly to those for BD security [15]. Moreover, leaks can consist of not only secrets but also actions, allowing relative security to synchronize not only the produced secrets but also the actions (attacker inputs). While in this paper we will focus on secret-based leaks (as in our attacker models), next we give an example that explores this action-extended alternative—which incidentally will tap into an interesting connection with related work.

Example 6 Consider the following system model instantiations $\mathcal{SM}_{\mathsf{van}}$ and $\mathcal{SM}_{\mathsf{opt}}$:

- $\mathcal{SM}_{\mathsf{van}}$: $\mathsf{State}_{\mathsf{van}} = (\mathbb{N}^3)$ (i.e., triples of natural numbers); $\mathsf{istate}_{\mathsf{van}}(i, m, n)$ holds iff $i = 1$; and vtransitions consist of $(1, m, n) \Rightarrow (3, m, m)$ for any m, n.
- $\mathcal{SM}_{\mathsf{opt}}$: $\mathsf{State}_{\mathsf{opt}} = \mathsf{State}_{\mathsf{van}}$; $\mathsf{istate}_{\mathsf{opt}}(i, m, n)$ holds iff $\mathsf{istate}_{\mathsf{van}}(i, m, n)$ or $i = 2$; and otransitions extend on the vtransitions to also allow $(2, m, n) \Rightarrow (3, m, m)$.

Hence, we have that $\mathsf{Trace}_{\mathsf{van}}$ consists of two-state traces of the form $(1, m, n)\,(3, m, m)$, and $\mathsf{Trace}_{\mathsf{opt}}$ similarly only has two-state traces, but they may also take the form $(2, m, n)\,(3, m, m)$.

Now consider two alternative vanilla leakage models $\mathcal{LM}_{\mathsf{van}}$ and $\mathcal{LM}'_{\mathsf{van}}$. Both are based on a secrecy and interaction infrastructure (as with attacker models), namely $\mathsf{Sec}_{\mathsf{van}} = \mathsf{Act}_{\mathsf{van}} = \mathsf{Obs}_{\mathsf{van}} = \mathbb{N}$ where given a trace in $\mathsf{Trace}_{\mathsf{van}}$, the only secret is defined as the second component of the initial state ($\mathsf{S}_{\mathsf{van}}((i_1, m_1, n_1)\,(i_2, m_2, n_2)) = m_1$), the action is the first component of the initial state ($\mathsf{A}_{\mathsf{van}}((i_1, m_1, n_1)\,(i_2, m_2, n_2)) = i_1$), and the observation is the third component of the final state ($\mathsf{O}_{\mathsf{van}}((i_1, m_1, n_1)\,(i_2, m_2, n_2)) = n_2$). Then:

- $\mathcal{LM}_{\mathsf{van}}$ is defined as an attacker model (i.e., $\mathcal{AM}_{\mathsf{van}}$) where $\mathsf{Leak}_{\mathsf{van}} = \mathsf{Sec}^*_{\mathsf{van}} \times \mathsf{Sec}^*_{\mathsf{van}}$ and $\mathsf{leakVia}_{\mathsf{van}}(\pi_1, \pi_2, (\sigma_1, \sigma_2))$ says $\mathsf{S}_{\mathsf{van}}(\pi_1) = \sigma_1 \wedge \mathsf{S}_{\mathsf{van}}(\pi_2) = \sigma_2 \wedge \mathsf{A}_{\mathsf{van}}(\pi_1) = \mathsf{A}_{\mathsf{van}}(\pi_2) \wedge \mathsf{O}_{\mathsf{van}}(\pi_1) \neq \mathsf{O}_{\mathsf{van}}(\pi_2)$.
- $\mathcal{LM}'_{\mathsf{van}}$ is defined as an "action-extended attacker model", namely $\mathsf{Leak}'_{\mathsf{van}} = \mathsf{Sec}^*_{\mathsf{van}} \times \mathsf{Sec}^*_{\mathsf{van}} \times \mathsf{Act}^*_{\mathsf{van}}$ and $\mathsf{leakVia}'_{\mathsf{van}}(\pi_1, \pi_2, (\sigma_1, \sigma_2, \alpha))$ says $\mathsf{S}_{\mathsf{van}}(\pi_1) = \sigma_1 \wedge \mathsf{S}_{\mathsf{van}}(\pi_2) = \sigma_2 \wedge \mathsf{A}_{\mathsf{van}}(\pi_1) = \mathsf{A}_{\mathsf{van}}(\pi_2) = \alpha \wedge \mathsf{O}_{\mathsf{van}}(\pi_1) \neq \mathsf{O}_{\mathsf{van}}(\pi_2)$.

The optimization-enhanced leakage models $\mathcal{LM}_{\mathsf{opt}}$ and $\mathcal{LM}'_{\mathsf{opt}}$ are defined in the same way, using traces from $\mathsf{Trace}_{\mathsf{opt}}$.

Hence, for the system models $\mathcal{SM}_{\mathsf{van}}$ and $\mathcal{SM}_{\mathsf{opt}}$:

- Relative security holds w.r.t. the leakage models $\mathcal{LM}_{\mathsf{van}}$ and $\mathcal{LM}_{\mathsf{opt}}$ (i.e., w.r.t. the attacker models $\mathcal{AM}_{\mathsf{van}}$ and $\mathcal{AM}_{\mathsf{opt}}$). Indeed, following the (★) formulation of relative security for attacker models (Def. 5), let $\pi_1, \pi_2 \in \mathsf{Trace}_{\mathsf{opt}}$ be such that $\mathsf{A}_{\mathsf{opt}}(\pi_1) = \mathsf{A}_{\mathsf{opt}}(\pi_2)$ and $\mathsf{O}_{\mathsf{opt}}(\pi_1) \neq \mathsf{O}_{\mathsf{opt}}(\pi_2)$. Then necessarily $\pi_1 = (i, m_1, n_1)\,(3, m_1, m_1)$ and $\pi_2 = (i, m_2, n_2)\,(3, m_2, m_2)$, where $i \in \{1, 2\}$ and $m_1 \neq m_2$. Taking $\hat{\pi}_1 = (1, m_1, n_1)\,(3, m_1, m_1)$ and $\hat{\pi}_2 = (1, m_2, n_2)\,(3, m_2, m_2)$, we have the desired (in)equalities:
  - $\mathsf{S}_{\mathsf{van}}(\hat{\pi}_1) = m_1 = \mathsf{S}_{\mathsf{van}}(\pi_1)$, $\mathsf{S}_{\mathsf{van}}(\hat{\pi}_2) = m_2 = \mathsf{S}_{\mathsf{van}}(\pi_2)$,
  - $\mathsf{A}_{\mathsf{van}}(\hat{\pi}_1) = 1 = \mathsf{A}_{\mathsf{van}}(\hat{\pi}_2)$,
  - $\mathsf{O}_{\mathsf{van}}(\hat{\pi}_1) = m_1 \neq m_2 = \mathsf{O}_{\mathsf{van}}(\hat{\pi}_2)$.
- However, relative security fails w.r.t. the leakage models $\mathcal{LM}'_{\mathsf{van}}$ and $\mathcal{LM}'_{\mathsf{opt}}$. Indeed, here relative security becomes:

$$
\begin{aligned}
&\forall \pi_1, \pi_2 \in \mathsf{Trace}_{\mathsf{opt}}.\ \mathsf{A}_{\mathsf{opt}}(\pi_1) = \mathsf{A}_{\mathsf{opt}}(\pi_2) \ \wedge\ \mathsf{O}_{\mathsf{opt}}(\pi_1) \neq \mathsf{O}_{\mathsf{opt}}(\pi_2)\\
&\longrightarrow\\
&\exists \hat{\pi}_1, \hat{\pi}_2 \in \mathsf{Trace}_{\mathsf{van}}.\\
&\qquad \mathsf{S}_{\mathsf{van}}(\hat{\pi}_1) = \mathsf{S}_{\mathsf{van}}(\pi_1) \ \wedge\ \mathsf{S}_{\mathsf{van}}(\hat{\pi}_2) = \mathsf{S}_{\mathsf{van}}(\pi_2) \ \wedge\\
&\qquad \mathsf{A}_{\mathsf{van}}(\hat{\pi}_1) = \mathsf{A}_{\mathsf{van}}(\hat{\pi}_2) = \mathsf{A}_{\mathsf{opt}}(\pi_1) \ \wedge\ \mathsf{O}_{\mathsf{van}}(\hat{\pi}_1) \neq \mathsf{O}_{\mathsf{van}}(\hat{\pi}_2)
\end{aligned}
$$

  To show that this fails, we take $\pi_1 = (2, m_1, n_1)\,(3, m_1, m_1)$ and $\pi_2 = (2, m_2, n_2)\,(3, m_2, m_2)$ where $m_1 \neq m_2$. Thus, we have $\mathsf{A}_{\mathsf{opt}}(\pi_1) = 2 = \mathsf{A}_{\mathsf{opt}}(\pi_2)$ and $\mathsf{O}_{\mathsf{opt}}(\pi_1) = m_1 \neq m_2 = \mathsf{O}_{\mathsf{opt}}(\pi_2)$. Because $\mathsf{A}_{\mathsf{opt}}(\pi_1) = 2$, there are no traces $\hat{\pi}_1, \hat{\pi}_2 \in \mathsf{Trace}_{\mathsf{van}}$ satisfying $\mathsf{A}_{\mathsf{van}}(\hat{\pi}_1) = \mathsf{A}_{\mathsf{van}}(\hat{\pi}_2) = \mathsf{A}_{\mathsf{opt}}(\pi_1)$.

Intuitively, both systems reveal the secret by copying it into the output, but in the vanilla system this happens only when the attacker takes action 1, whereas in the optimization-enhanced system this also happens when the attacker takes action 2. Relative security based on our attacker models deems this secure, because a leak (given by a pair of sequences of secrets, here singleton sequences (m_1, m_2)) in the optimization-enhanced system can be reproduced in the vanilla system, albeit triggered by different actions; but in the "action-extended attacker models", relative security requires coincidence on the leak-triggering actions as well. □

(Non)determinism. While we formulated relative security for possibly nondeterministic systems, there is a caveat to be considered when deploying it to concrete systems. This is *not* due to the way relative security works across systems, but instead with how the notion of a leak is defined separately for each system. Namely, in an attacker model, a leak is given by pair of traces π_1, π_2 such that $\mathsf{A}(\pi_1) = \mathsf{A}(\pi_2)$ and $\mathsf{O}(\pi_1) \neq \mathsf{O}(\pi_2)$. This follows the pattern of Goguen-Meseguer noninterference [16], which was originally designed for deterministic systems, and is guaranteed to be meaningful for such systems. On the other hand, when switching to nondeterministic systems, one must make sure not to count as leaks the "pseudo-leaks" that stem from nondeterminism. For example, Zdancewic and Myers's low observational determinism [17] is a generalization of noninterference that is meaningfully applied to nondeterministic systems of concurrent threads provided they are race-free. In our framework, the absence of pseudo-leaks for an attacker model can be expressed as follows: Given any two traces starting in *the same* initial state, if they have the same actions then they yield the same observations. Formally, if Trace_s denotes the set of traces starting in state s, this is expressed as follows: $\forall s, \pi_1, \pi_2.\ \mathsf{istate}\ s \wedge \{\pi_1, \pi_2\} \subseteq \mathsf{Trace}_s \wedge \mathsf{A}(\pi_1) = \mathsf{A}(\pi_2) \rightarrow \mathsf{O}(\pi_1) = \mathsf{O}(\pi_2)$. In particular, this holds true for deterministic systems, where π_1 and π_2 turn out to be the same trace. All our instantiations of the relative security framework, including the main one to speculative execution systems described in §7 and §8, employ deterministic system models.

3.4 State-wise Attacker Models

Next, we will consider an assumption on attacker models that makes them even less abstract. Namely, we make S, A and O more concrete by assuming that they operate "state-wise", i.e., the secrets, actions and observations are produced locally from each state of a trace.

Def 7 An attacker model $\mathcal{AM} = (\mathsf{Sec}, \mathsf{S}, \mathsf{Obs}, \mathsf{O}, \mathsf{Act}, \mathsf{A})$ for a given system model $\mathcal{SM} = (\mathsf{State}, \mathsf{istate}, \Rightarrow)$, is said to be *state-wise* when there exist the predicates and functions $\mathsf{isSec} : \mathsf{State} \rightarrow \mathsf{Bool}, \mathsf{getSec} : \mathsf{State} \rightarrow \mathsf{Sec}, \mathsf{isInt} : \mathsf{State} \rightarrow \mathsf{Bool}, \mathsf{getObs} : \mathsf{State} \rightarrow \mathsf{Obs}$, and $\mathsf{getAct} : \mathsf{State} \rightarrow \mathsf{Act}$ that define the functions S, O and A state-wise as follows. For any trace $\pi = s_0\, s_1 \ldots$:

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

What is being required above is that the trace functions S, O and A are defined by a "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 and/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.

4 Formalizing Relative Security

Isabelle/HOL[5], henceforth referred to as Isabelle, is a proof assistant built on higher order logic [18]. We use Isabelle to mechanize the relative security framework (§3). This section begins by summarizing some necessary Isabelle background for the remainder of the paper (§4.1), and the overall theory structure of the abstract formalization (§4.2), before providing details on the mechanization process (§4.3).

4.1 Isabelle Background

Our mechanization relies heavily on Isabelle's module system, known as *locales* [19, 20]. A locale fixes a combination of parameters (types and constants) and assumptions. Local definitions and proofs relative to these can then be written within a locale context. All local theorems and definitions are also accessible from outside the locale [21], however are viewed as **1)** polymorphic in that locale's fixed types, **2)** universally quantified over that locale's constants, and **3)** conditioned by that locale's assumptions.

Locales provide a flexible and extensible inheritance hierarchy, which is vital in our use case. An existing locale can be extended with new parameters and/or assumptions in a new locale definition (direct inheritance), or two locales, L and L', can be related to one another via a **sublocale** declaration, $L' \leq L$ (indirect inheritance). An **interpretation** at the top level of an Isabelle theory *interprets* a locale by instantiating its parameters. A **sublocale** can be viewed as a relative interpretation, where L' may be the more concrete version of the general locale L. For both **sublocale**'s and **interpretations**, the locale's assumptions must be verified via a proof with respect to the instantiated (or more concrete) parameters.

4.2 Theory Structure

Fig. 2 shows the theory structure of our Isabelle session for formalizing relative security [6]. Each theory contains numerous definitions and proofs grouped by topic. Theory names are mostly self-explanatory, e.g. `Relative_Security`. The suffix "fin" refers to the finitary (finite-trace) versions of the concepts. Note the bottom two branches of this tree refer to the unwinding proof theory that we develop in (§5).

Our development has limited dependencies on pre-existing Isabelle libraries (represented by square brackets in Fig. 2, typically encompassing multiple theories). Beyond the Isabelle distribution, the only library of note is the AFP entry on coinduction [22] (`[Coinductive]`), which includes formal theories on coinductive (lazy) lists. Our own work begins with a substantial library of general extensions to this coinductive list development (1K LOC), published as a separate AFP entry [23]. The `Trivia` theory then includes further background formalization work on filtermaps for lists and lazy lists, as specifically required for this paper's use case. The remaining theories are focused on relative security and unwinding, consisting of about 11K LOC in total.

4.3 A Locale Approach to Formalizing Relative Security

In (§3) we gradually refined our definition of relative security from a very abstract definition to a slightly more concrete one. Many of the definitions are closely related, e.g., an attacker model is defined with respect to a system model. Furthermore, to use these definitions in a

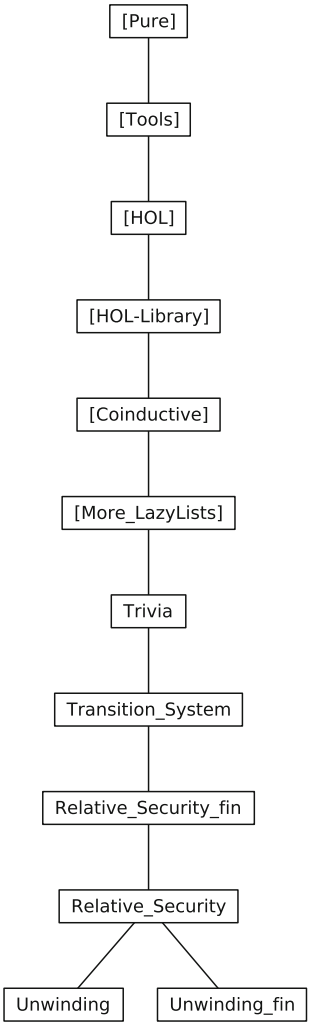


Fig. 2 Theory structure of our abstract Isabelle mechanization.

concrete setting, we know we'll need to instantiate the parameters of each. Such a refinement based approach naturally leads towards a locale-centric strategy for formalizing relative security.

Fig. 3 provides an overview of the locale hierarchy for our definitions. There is a symmetry in this tree due to the duality between the general and finitary definitions of relative security. The main difference between these definitions is the use of lists (for finitary traces) and lazy lists (for possibly infinite traces). For simplicity, we focus on the branches corresponding to the general definitions for the remainder of this section.

System models. The *Transition_System* theory is used to formalize system models based on Def. 1. We begin by defining a simple *Transition_System* locale, which simply **fixes** the initial state predicate (*istate*) and transition relation as parameters (e.g. *validTrans*).

```
locale Transition_System =
  fixes istate :: "'state ⇒ bool"
    and validTrans :: "'trans ⇒ bool"
```

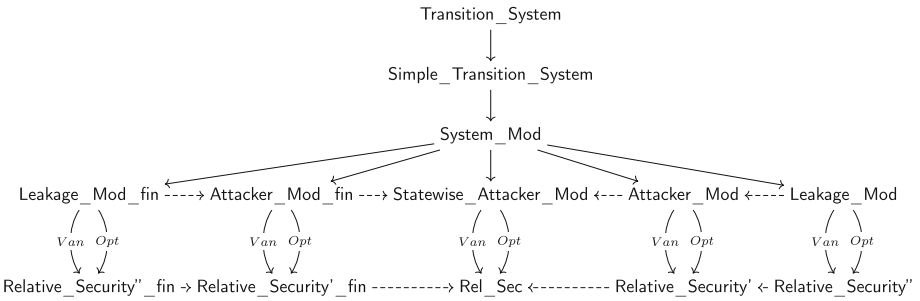


Fig. 3 Relative security locale inheritance structure. Solid arrows represent *direct* inheritance, and dashed arrows *indirect* inheritance, i.e. via sublocales.

```
  and srcOf :: "'trans ⇒ 'state"
  and tgtOf :: "'trans ⇒ 'state"
```

Note that the set of states from the original definition is defined implicitly using the (`'state`) type.

In Def. 1, a transition is simply represented as pairs of states. Hence, we introduce the `Simple_Transition_System` locale which builds on `Transition_System` however instantiates the generic `'trans` type with (`'state × 'state`) and specifies the `srcOf` and `tgtOf` parameters in terms of the `validTrans` relation. This removes the need to specify redundant information when using the locale (via inheritance or instantiation) for reasoning about transitions represented this way.

To finish formalizing Def. 1 we define the `System_Mod` locale, which extends the `Simple_Transition_System` locale by adding a parameter and an assumption for reasoning on finality. In the Isabelle snippet below, note the addition of an **assumes** statement used to state the locale assumption, and **for** keyword to specify the type of the inherited parameters.

```
locale System_Mod =   Simple_Transition_System istate validTrans
  for istate :: "'state ⇒ bool"
  and validTrans :: "'state × 'state  ⇒ bool"
  +
  fixes final :: "'state ⇒ bool"
  assumes final_def: "final s1 ⟷ (∀s2. ¬ validTrans (s1,s2))"
```

The locales above set up many useful definitions and lemmas within their local contexts to support reasoning over states (e.g. reachability properties) and valid traces (both finite and possibly infinite). Most of these lemmas are located in the more general transition system locale, with the `System_Mod` context only needed for reasoning specifically on *completed* traces, i.e. traces which require the `final` parameter and assumption.

Leakage and attacker models. Recall that the most abstract definition of relative security uses *leakage models* (Def. 2), which we formalize within the `Leakage_Mod` locale by extending the `System_Model` locale. In particular, we fix the `lleakVia` parameter—a function representing the concept of two traces exhibiting a leak, with the leak set formalized via the `'leak` type.

```
locale Leakage_Mod = System_Mod istate validTrans final
  for istate :: "'state ⇒ bool"
 and validTrans :: "'state × 'state ⇒ bool"
 and final :: "'state ⇒ bool"
```

```
  +
  and lleakVia :: "'state llist ⇒ 'state llist ⇒ 'leak ⇒ bool"
```

The more concrete `Attacker_Mod` locale similarly extends `System_Mod`, additionally introducing the `S`, `A`, and `O` parameters, in line with the original definition (Def. 4).

```
locale Attacker_Mod = System_Mod istate validTrans final
  for istate :: "'state ⇒ bool"
 and validTrans :: "'state × 'state ⇒ bool"
 and final :: "'state ⇒ bool"
  +
  fixes S :: "'state llist ⇒ 'secret llist"
  and A :: "'state ltrace ⇒ 'act llist"
  and O :: "'state ltrace ⇒ 'obs llist"
```

However, unlike `Leakage_Mod` it does *not* fix `lleakVia` as a parameter, as the leakVia function is now defined in terms of the secrets, observations, and actions introduced by our new parameters. As such, we define a function concretely in the context of the locale which uses the respective locale parameters:

```
fun lleakVia :: "'state llist ⇒ 'state llist ⇒
    'secret llist × 'secret llist ⇒ bool" where
  "lleakVia tr tr' (sl,sl') = (S tr = sl ∧ S tr' = sl' ∧ A tr = A tr' ∧
    O tr ≠ O tr')"
```

We must also ensure that the attacker model for a given SM induces a leakage model for that SM. In other words, any lemmas proven in the more abstract leakage model should be usable in the context of an attacker model as well. This can be achieved via sublocales (i.e., indirect inheritance). Below, the **where** keyword instantiates the abstract parameter `lleakvia` from `Leakage_Mod` with the concrete function defined in `Attacker_Mod` whenever reasoning in the latter's context.

```
sublocale Attacker_Mod < Leakage_Mod
  where lleakVia = lleakVia
  by standard
```

Finally, we consider the least abstract state-wise attacker models (Def. 7).

```
locale Statewise_Attacker_Mod = System_Mod istate validTrans final
  for istate :: "'state ⇒ bool" and validTrans :: "'state × 'state ⇒
bool"
 and final :: "'state ⇒ bool"
  +
  fixes   isSec :: "'state ⇒ bool" and getSec :: "'state ⇒ 'secret"
   and  isInt :: "'state ⇒ bool" and getInt :: "'state ⇒ 'act × 'obs"
  assumes final_not_isInt: "⋀s. final s ⟹ ¬ isInt s"
  and final_not_isSec: "⋀s. final s ⟹ ¬ isSec s"
```

This once again directly inherits from the `System_Mod` locale and extends it by fixing the predicates `isSec` and `isInt`, and functions `getSec` and `getInt` as new locale parameters. Note that here, we do not have the `S`, `O`, and `A` parameters of `Attacker_Mod` in the locale definition, since they are now defined concretely using filtermap. This leads to two important side-effects.

First, none of the parameters differentiate between finite and possibly infinite traces, and hence `Statewise_Attacker_Mod` supports formalization of both cases via appropriate definitions within the locale. For example, the observation function for the possibly infinite case using lazy lists and lazy traces (`llist` and `ltrace`) is given below.

```
definition lO :: "'state ltrace ⇒ 'obs llist" where
  "lO tr ≡ lfiltermap isInt getObs (lbutlast tr)"
```

Second, we obtain a further usecase for locales. The *Trivia* theory sets up a locale *LfiltermapBL* (and *filtermapBL* in the finite case), which represent contexts that apply a filtermap to all but the last element of a list. Our definitions of S, O, and A satisfy this locale's parameters and assumptions, resulting in the following sublocale declaration within the *Statewise_Attacker_Mod* context:

```
sublocale lO: LfiltermapBL isInt getObs lO
  apply standard unfolding lO_def ..
```

This allows us to use *lO.l* to refer to lemmas (instantiated with the filtermap defined by *lO*) that were simply proved within the more general filtermap locale context. We saw improvements in modularity and proof automation using this approach, avoiding the need to consistently unwrap definitions if the proofs had been done without locales.

Relative Security Models. The relative security locales (bottom of Fig. 3) are defined by inheriting two labelled models. One for the vanilla system, *Van*, and another for the optimized system, *Opt*. For example, the locale below defines the parameters for our "less abstract" definition of relative security by directly inheriting from two instances of the *Attacker_Mod* locale.

```
locale Relative_Security' =
    Van: Attacker_Mod istateV validTransV finalV SV AV OV
  +
    Opt: Attacker_Mod istateO validTransO finalO SO AO OO
    for validTransV :: "'stateV × 'stateV ⇒ bool"
    and istateV :: "'stateV ⇒ bool" and finalV :: "'stateV ⇒ bool"
    and SV :: "'stateV llist ⇒ 'secret llist"
    and AV :: "'stateV ltrace ⇒ 'actV llist"
    and OV :: "'stateV ltrace ⇒ 'obsV llist"

    and validTransO :: "'stateO × 'stateO ⇒ bool"
    and istateO :: "'stateO ⇒ bool" and finalO :: "'stateO ⇒ bool"
    and SO :: "'stateO llist ⇒ 'secret llist"
    and AO :: "'stateO ltrace ⇒ 'actO llist"
    and OO :: "'stateO ltrace ⇒ 'obsO llist"
    and corrState :: "'stateV ⇒ 'stateO ⇒ bool"
```

Note that the locale declaration above does not include any new assumptions, with the majority of the *for* statement in each relative security definition simply setting up useful names for the types of each inherited parameter. The only new addition is the *corrState* parameter, which is a predicate that holds iff vstate and ostate are corresponding states. This always evaluates to true in the context of our motivating examples from Sec. 2, however enables more flexibility in the overall model.

The condition for relative security is left for a definition inside the locale context: *rsecure* in the finite case and *lrsecure* for the general case. The definition is located in the most abstract *Relative_Security"* locales, corresponding to Def. 3. In the *Relative_Security'* locales we prove a lemma showing this definition is equivalent to our attacker model reformulation given in Def. 5. The most concrete *Rel_Sec* locale (building on state-wise attacker models) inherits both the *rsecure* and *lrsecure* definitions—note that neither implies the other.

This locale structure enables us to reason at a more abstract level when possible, and still use the results of that reasoning at a more concrete level. It additionally validates the suspected relationships between the definitions in (§3).

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 (§3.4). We first discuss design challenges stemming from the four-trace constraint system specific to relative security (§5.1), then converge to a definition of an unwinding relation (§5.2), additionally offering a finite-trace variant that allows for simpler conditions. This is followed by some toy examples (§5.3) to provide further intuition behind the definition's usage and a brief discussion of the definition's formalization (§5.4). This then culminates with the formal statement of soundness for the proof method, i.e., that unwinding indeed ensures relative security (§5.5). We also introduce a compositional variant of unwinding, which allows us to decompose the proof in different unwinding relations (§5.6). Finally, we study the dual problem of incrementally *disproving* relative security. This leads to a proof method that we call secret-directed unwinding because it shows the impossibility of saturating given sequences of secrets (§5.7). Formalization was essential to developing the soundness proofs of our unwinding (dis)proof methods, thus we touch on interesting aspects of the mechanization alongside discussing the intuition behind these proofs (§5.8). We also discuss the incompleteness of the (dis)proof methods (§5.9).

5.1 Design Aspects

Unwinding is a (bi)simulation-like [24] method specialised in proving noninterference and related two-trace properties [4, 25]. 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 pair of otraces π_1 and π_2 as well as the pair of counterpart vtraces $\hat{\pi}_1$ and $\hat{\pi}_2$ correspond to pairs 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 require a mechanism for building the vtraces $\hat{\pi}_1$ and $\hat{\pi}_2$ incrementally from the otraces π_1 and π_2 such that they create the same leak, i.e., take the same actions *and* generate the same secrets *yet* produce different observations. During the unwinding, we must 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, (R1) will be *assumed*, whereas (R2) and (R3) must be *guaranteed*. Moreover, assuming/guaranteeing that two generated sequences *stay equal* (like for the sequences of *actions* in (R1)–(R3)) has a different flavor 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:

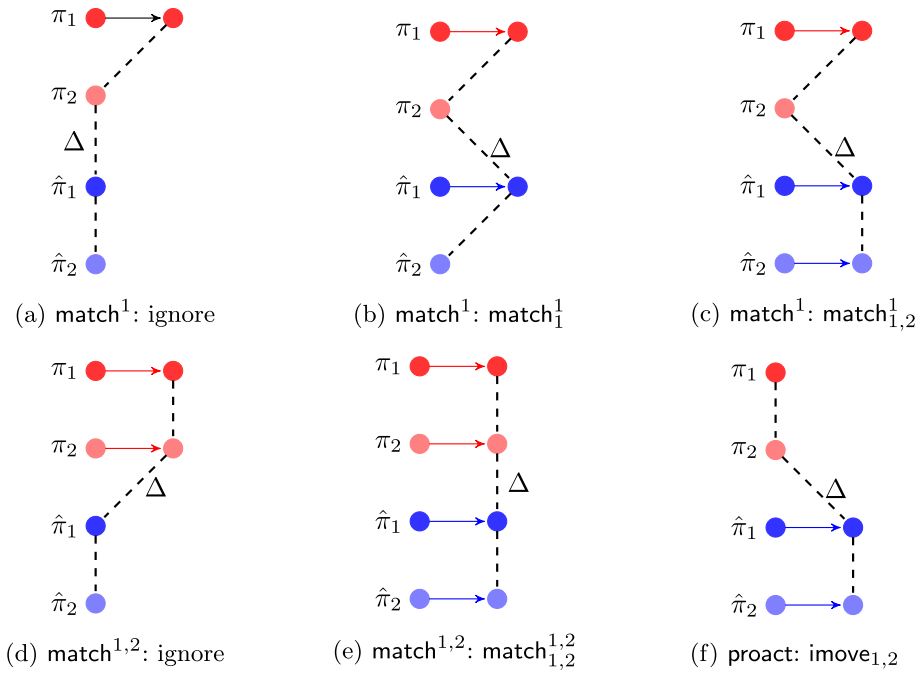


Fig. 4 Some transition matching and proactive move patterns. The subfigure labels show the relevant top-level predicate, e.g, match^1, followed (after semi-colon) by information about the relevant disjunct in the definition of this predicate. For example, match$^1_1$ is the predicate appearing in the second disjunct in the definition of match^1, and "ignore" refers to the first disjunct (which corresponds to ignoring the transition(s)). The red bullets indicate the current states of the otraces, and the blue bullets those of the counterpart vtraces. The arrows indicate the transitions taken under each specific move; for example, in subfigure (e) both otraces take transitions, and these are matched by transitions from both vtraces.

1. 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; and
2. 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. 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. A visual representation of the key steps and some of these variations is available in Fig. 4a-4e. The side-conditions when applying these matching mechanisms ensure that we maintain "(R1) implies (R2) and (R3)".

In the rest of the paper, we will refer to the conditions (R1)–(R3) more intuitively, as follows.

- 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 of $\hat{\pi}_1$ and $\hat{\pi}_2$ (eventually) being different, provided that they are also different for π_1 and π_2.

- A *secrecy contract*, corresponding to "(R1) implies (R3)", 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, as shown in Fig. 4f. 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* (expanded on later in this section).

Next, we will give the definition of unwinding, which is very technical. While reading the definition and the intuitive explanations of its various components, the reader may prefer to look at the subsequent "toy examples" section (§5.3); in particular, Examples 11 and 12 from there illustrate the roles (and the necessity) of the numeric timer parameters.

5.2 Definition of Unwinding

Def. 8 gives the formal definition. We let $\mathsf{Status} = \{\mathsf{eq}, \mathsf{diff}\}$, where eq signifies equality and diff signifies difference/divergence. Let $\mathbb{N}_\infty = \mathbb{N} \cup \{\infty\}$ be 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 8 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 its subpredicates $\mathsf{match}(\Delta)$) and $\mathsf{proact}(\Delta)$, cover the many cases of the aforementioned reactive and proactive components of unwinding. The full definitions are presented in Figs. 5 and 6 respectively, and are discussed in detail below. The definitions in these figures use the parameters of relative security either directly (e.g., isSec and isInt) or via the following auxiliary functions eqSec, eqAct and newStat:

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

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'.

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

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

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

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

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

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

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

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

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

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

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

Fig. 5 Definition of $\mathsf{react}(\Delta)$ – the reactive component of unwinding

$$
\begin{aligned}
&\mathsf{proact}(\Delta)\ v\,(v_1, v_2)(s_1, s_2, st)\ (\hat{s}_1, \hat{s}_2, \hat{st}) \doteq \\
&\quad \neg\,\mathsf{isSec}(s_1) \wedge \neg\,\mathsf{isInt}(s_1) \wedge \mathsf{imove}_1(\Delta)\ v\,(v_1, v_2)(s_1, s_2, st)\ (\hat{s}_1, \hat{s}_2, \hat{st}) \vee \\
&\quad \neg\,\mathsf{isSec}(s_1) \wedge \neg\,\mathsf{isInt}(s_1) \wedge \mathsf{imove}_2(\Delta)\ v\,(v_1, v_2)(s_1, s_2, st)\ (\hat{s}_1, \hat{s}_2, \hat{st}) \vee \\
&\quad \neg\,\mathsf{isSec}(s_1) \wedge \neg\,\mathsf{isSec}(s_2) \wedge \mathsf{eqAct}(s_1, s_2) \wedge \mathsf{imove}_{1,2}(\Delta)\ v\,(v_1, v_2)(s_1, s_2, st)\ (\hat{s}_1, \hat{s}_2, \hat{st}) \\
&\mathsf{imove}_1(\Delta)\ v\,(v_1, v_2)(s_1, s_2, st)\ (\hat{s}_1, \hat{s}_2, \hat{st}) \doteq \\
&\quad \exists \hat{s}_1' \in \mathsf{State}_{\mathsf{van}}.\ \hat{s}_1 \Rightarrow \hat{s}_1' \wedge \Delta\ v\,(v_1, v_2)(s_1, s_2, st)\ (\hat{s}_1', \hat{s}_2, \hat{st}) \\
&\mathsf{imove}_2(\Delta)\ v\,(v_1, v_2)(s_1, s_2, st)\ (\hat{s}_1, \hat{s}_2, \hat{st}) \doteq \\
&\quad \exists \hat{s}_2' \in \mathsf{State}_{\mathsf{van}}.\ \hat{s}_2 \Rightarrow \hat{s}_2' \wedge \Delta\ v\,(v_1, v_2)(s_1, s_2, st)\ (\hat{s}_1, \hat{s}_2', \hat{st}) \\
&\mathsf{imove}_{1,2}(\Delta)\ v\,(v_1, v_2)(s_1, s_2, st)\ (\hat{s}_1, \hat{s}_2, \hat{st}) \doteq \\
&\quad \text{let } \hat{st}' = \mathsf{newStat}(\hat{st}, \hat{s}_1, \hat{s}_2) \text{ in} \\
&\qquad\qquad \exists \hat{s}_1', \hat{s}_2' \in \mathsf{State}_{\mathsf{van}}.\ \hat{s}_1 \Rightarrow \hat{s}_1' \wedge \hat{s}_2 \Rightarrow \hat{s}_2' \wedge \Delta\ v\,(v_1, v_2)(s_1, s_2, st)\ (\hat{s}_1', \hat{s}_2', \hat{st}')
\end{aligned}
$$

Fig. 6 Definition of $\mathsf{proact}(\Delta)$ – the proactive component of unwinding.

A note on notation: in Figs. 5 and 6, 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 $\mathsf{match}^{1,2}(\Delta)$. 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 $\mathsf{match}_1^{1,2}(\Delta)$, $\mathsf{match}_2^{1,2}(\Delta)$ and $\mathsf{match}_{1,2}^{1,2}(\Delta)$. We have further annotated Fig. 4 with this notation in the relevant steps.

Reactive growth. Predicate $\mathsf{react}(\Delta)$ (see Fig. 5) 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 right-hand side of the implication from $s_1 \Rightarrow s_1'$:

1) either ignoring the transition, 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 $\mathsf{match}_{1,2}^1(\Delta)$, 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 (does not 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 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. 6) 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 are 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. This 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. 8 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.

Discussion on finitary unwinding. We first focus on finitary unwinding (Def. 9), ignoring everything highlighted in gray—this will be sufficient for finitary relative security.

Def 9 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. 8 holds when ignoring the two $\mathbb{N}_\infty$ arguments highlighted in gray (including in Figs. 5 and 6).

Typically, a finitary unwinding relation Δ encodes:

- the current states of the otraces, s_1 and s_2 (call them "ostates"),
- the observation divergence status, st, of s_1 and s_2 (call it "ostatus"),
- the current states $\hat{s}_1$ and $\hat{s}_2$ (call them "vstates") and the status, $\hat{st}$ (call it "vstatus"), of the vtraces which are being constructed,
- a timer parameter $v \in \mathbb{N}_\infty$ (for bounding proactive growth, explained later).

Def. 9 ensures that Δ can support the secure growth of the vtraces, reactively or proactively.

Upgrading to general unwinding. In the presence of infinite traces, there are more opportunities for "filibustering" in addition to the one discussed above coming from unbounded proactive moves (by the vtraces), for example from the unbalanced reactive moves where one of the vtraces $\hat{\pi}_i$ grows indefinitely and "starves" the other one, or from the otraces growing indefinitely while their vtrace counterparts are starving. This would allow us to successfully play the unwinding game without proving relative security, i.e., without proving that the constructed $\hat{\pi}_1$ and $\hat{\pi}_2$ have the same observations or that $\hat{\pi}_i$ and π_i have the same secrets (more precisely, that $\hat{\pi}_i$ does not have additional secrets that have not been explored). To counter this, we use the additional timers v_1 and v_2 highlighted in Def. 8 and Figs. 5 and 6. 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. 5 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 Unwinding Toy Examples

While our running examples from §2 will be analyzed in the context of more elaborate attacker models (in §8), next we show some toy examples (similar to Example 6) that illustrate our definition of unwinding and the discussed design decisions behind it. As seen in Def. 8, verifying that a relation Δ is an unwinding goes as follows: We assume that some tuples are related by Δ, and show that after suitably constrained (possibly matched) transitions the results are again related by Δ. We will refer to this verification process as a "proof by unwinding". In practice, Δ will be a union (disjunction) of certain configuration patterns, and a proof by unwinding will show how we can "move" between these patterns, or "stay" within a pattern.

Example 10 Consider the following system models and attacker models:

- $\mathcal{SM}_{\mathsf{van}}$: $\mathsf{State}_{\mathsf{van}} = (\mathbb{N}^3)_\perp$ (i.e., either triples of natural numbers or $\perp$); $\mathsf{istate}_{\mathsf{van}}((i,m,n))$ iff $i = 1 \wedge n = 0$; and vtransitions are $(1,m,0) \Rightarrow (3,m,0) \Rightarrow (3,m,0) \Rightarrow (4,m,m) \Rightarrow \perp$ for any m. $\mathsf{Trace}_{\mathsf{van}}$ thus consists of finite traces of the form $(1,m,0)\,(3,m,0)^k\,(4,m,m)\,\perp$ where $k \in \mathbb{N} \setminus \{0\}$ and infinite traces of the form $(1,m,0)\,(3,m,0)^\infty$.
- $\mathcal{SM}_{\mathsf{opt}}$: $\mathsf{State}_{\mathsf{opt}} = \mathsf{State}_{\mathsf{van}}$; $\mathsf{istate}_{\mathsf{opt}}((i,m,n))$ iff $\mathsf{istate}_{\mathsf{van}}((i,m,n))$ or $i = 2 \wedge n = 0$; and the otransitions extend on the vtransitions to also allow $(2,m,0) \Rightarrow (4,m,m)$ for any m. $\mathsf{Trace}_{\mathsf{opt}}$ therefore includes $\mathsf{Trace}_{\mathsf{van}}$ as well as finite traces of the form $(2,m,0)\,(4,m,m)\,\perp$.
- $\mathcal{AM}_{\mathsf{van}}$ consists of $\mathsf{Sec}_{\mathsf{van}} = \mathsf{Act}_{\mathsf{van}} = \mathsf{Obs}_{\mathsf{van}} = \mathbb{N}$, and:
  - $\neg\,\mathsf{isSec}(\perp)$, $\mathsf{isSec}_{\mathsf{van}}(i,m,n) \leftrightarrow i \in \{1,2\}$, $\mathsf{getSec}_{\mathsf{van}}(i,m,n) = m$;
  - $\neg\,\mathsf{isInt}(\perp)$, $\mathsf{isInt}_{\mathsf{van}}(i,m,n) \leftrightarrow i \in \{1,2,4\}$, $\mathsf{getAct}_{\mathsf{van}}(i,m,n) = i$, $\mathsf{getObs}_{\mathsf{van}}(i,m,n) = n$.
- $\mathcal{AM}_{\mathsf{opt}}$ is given by the same predicates and functions as $\mathcal{AM}_{\mathsf{van}}$.

We have that relative security holds for these systems. Indeed, using a trace-based argument, since $\mathcal{AM}_{\mathsf{van}} = \mathcal{AM}_{\mathsf{opt}}$ and $\mathsf{Trace}_{\mathsf{van}} \subseteq \mathsf{Trace}_{\mathsf{opt}}$, the only possibly problematic leaks come from pairs of otraces in $\mathsf{Trace}_{\mathsf{opt}} \setminus \mathsf{Trace}_{\mathsf{van}}$, where action sequences are necessarily different as they start with 2 instead of 1. So let π_1, π_2 be two such otraces where $\mathsf{A}_{\mathsf{opt}}(\pi_1) = \mathsf{A}_{\mathsf{opt}}(\pi_2)$ and $\mathsf{O}_{\mathsf{opt}}(\pi_1) \neq \mathsf{O}_{\mathsf{opt}}(\pi_2)$. Then necessarily $\pi_1 = (2,m_1,0)\,(4,m_1,m_1)\,\perp$ and $\pi_2 = (2,m_2,0)\,(4,m_2,m_2)\,\perp$ for some m_1, m_2 such that $m_1 \neq m_2$, hence $\mathsf{A}_{\mathsf{opt}}(\pi_1) = 2\ 4 = \mathsf{A}_{\mathsf{opt}}(\pi_2)$ and $\mathsf{O}_{\mathsf{opt}}(\pi_1) = 0\ m_1 \neq 0\ m_2 = \mathsf{O}_{\mathsf{opt}}(\pi_2)$. We take the vtraces

$\hat{\pi}_1 = (1, m_1, 0)\ (3, m_1, 0)\ (4, m_1, m_1)\ \bot$ and $\hat{\pi}_2 = (1, m_2, 0)\ (3, m_2, 0)\ (4, m_2, m_2)\ \bot$, for which we have:

- $\mathsf{S}_{\mathsf{van}}(\hat{\pi}_1) = m_1 = \mathsf{S}_{\mathsf{opt}}(\pi_1), \mathsf{S}_{\mathsf{van}}(\hat{\pi}_2) = m_2 = \mathsf{S}_{\mathsf{opt}}(\pi_2)$,
- $\mathsf{A}_{\mathsf{van}}(\hat{\pi}_1) = 1\ 4 = \mathsf{A}_{\mathsf{van}}(\hat{\pi}_2), \mathsf{O}_{\mathsf{van}}(\hat{\pi}_1) = 0\ m_1 \neq 0\ m_2 = \mathsf{O}_{\mathsf{van}}(\hat{\pi}_2)$,

which proves relative security (in this only possibly problematic case).

To establish this instead via unwinding, we define $\Delta\ v\ (v_1, v_2)\ (s_1, s_2, st)\ (\hat{s}_1, \hat{s}_2, \hat{st})$ to say that $v_1 = v_2 = \infty$ and the disjunction of the following four statements holds:

i) $s_1 = s_2 = \hat{s}_1 = \hat{s}_2 \in \{(1, m, 0),\ (3, m, 0),\ (4, m, m)\}$ and $st = \hat{st} = \mathsf{eq}$;
ii) $s_1 = s_2 = (2, m, 0)$, $\hat{s}_1 = \hat{s}_2 = (1, m, 0)$ and $st = \hat{st} = \mathsf{eq}$;
iii) $s_1 = s_2 = (4, m, m)$, $\hat{s}_1 = \hat{s}_2 = (3, m, 0)$ and $st = \hat{st} = \mathsf{eq}$;
iv) $s_1 = s_2 = \hat{s}_1 = \hat{s}_2 = \bot$ and $st = \hat{st}$.

Here Δ effectively relates the transitions between states as follows: i) all the active states (i.e., with outward transitions) in $\mathcal{SM}_{\mathsf{van}}$ to themselves; ii) the additional active state in $\mathcal{SM}_{\mathsf{opt}}$ $((2, m, 0))$ to the state $(1, m, 0)$; iii) the state which $(2, m, 0)$ transits to in $\mathcal{SM}_{\mathsf{opt}}$ $((4, m, m))$ to that which $(1, m, 0)$ transits to in $\mathcal{SM}_{\mathsf{van}}$ $((3, m, 0))$; and iv) the final state $\bot$ with itself. Notably, in all except for (iv) the status divergence is eq (i.e., observations have not diverged), whereas in (iv) it may be eq or diff.

The "proof by unwinding" means verifying that Δ is indeed an unwinding. In the only critical part (essentially following $\pi_1, \pi_2, \hat{\pi}_1, \hat{\pi}_2$ from above)

the proof proceeds as follows:

- Assume we have $\Delta\ v\ (\infty, \infty)\ ((2, m_1, 0), (2, m_2, 0), \mathsf{eq})\ ((1, m_1, 0), (1, m_2, n_1), \mathsf{eq})$.
- We proceed by showing the react disjunct holds from the unwinding definition (Def. 8), i.e., we can move to react $\Delta\ (\infty, \infty)\ ((2, m_1, 0), (2, m_2, 0), \mathsf{eq})\ ((1, m_1, 0), (1, m_2, 0), \hat{st})$. This requires unfolding our react and matching definitions (from Fig. 5):
  - First, observe since $\mathsf{isInt}(2, m_1, 0)$ and $\mathsf{isInt}(2, m_2, 0)$ hold, the only non-vacuous conjunct to verify in the react definition is $\mathsf{match}^{1,2}\ \Delta\ (\infty, \infty)\ ((2, m_1, 0), (2, m_2, 0), \mathsf{eq})\ ((1, m_1, 0), (1, m_2, 0), \mathsf{eq})$.
  - Next observe, $\mathsf{eqSec}((2, m_i, 0), (1, m_i, 0))$ and $\mathsf{eqAct}((1, m_1, 0), (1, m_2, 0))$ hold. Thus we take the fourth disjunct in the $\mathsf{match}^{1,2}$ definition, and must show $\mathsf{match}^{1,2}_{1,2}\ \Delta\ (\infty, \infty)\ ((4, m_1, m_1), (4, m_2, m_2), \mathsf{eq})\ ((1, m_1, 0), (1, m_2, 0), \mathsf{eq})$.
    Note as the only possible otransitions are $(2, m_1, 0) \Rightarrow (4, m_1, m_1)$ and $(2, m_2, 0) \Rightarrow (4, m_2, m_2)$, this effectively fixes the states to consider in the universal quantification.
  - In the $\mathsf{match}^{1,2}_{1,2}$ definition, we instantiate the existential quantifiers with the vstates $(3, m_1, 0)$ and $(3, m_2, 0)$, for which we have $(1, m_1, 0) \Rightarrow (3, m_1, 0)$ and $(1, m_2, 0) \Rightarrow (3, m_2, 0)$; and, since the new observation divergence status (obtained by applying the $\mathsf{newStat}$ auxiliary function) is still eq, we reach $\Delta\ \infty\ (\infty, \infty)\ ((4, m_1, m_1), (4, m_2, m_2), \mathsf{eq})\ ((3, m_1, 0), (3, m_2, 0), \mathsf{eq})$.

  A short way to describe the above part is saying that so far, we moved:
  - from $\Delta\ v\ (\infty, \infty)\ ((2, m_1, 0), (2, m_2, 0), \mathsf{eq})\ ((1, m_1, 0), (1, m_2, 0), \mathsf{eq})$
  - to $\Delta\ \infty\ (\infty, \infty)\ ((4, m_1, m_1), (4, m_2, m_2), \mathsf{eq})\ ((3, m_1, 0), (3, m_2, 0), \mathsf{eq})$
  - via react–$\mathsf{match}^{1,2}$–$\mathsf{match}^{1,2}_{1,2}$.
- After a similar analysis as above
  we move:
  - from $\Delta\ \infty\ (\infty, \infty)\ ((4, m_1, m_1), (4, m_2, m_2), \mathsf{eq})\ ((3, m_1, 0), (3, m_2, 0), \mathsf{eq})$
  - to $\Delta\ 0\ (\infty, \infty)\ ((4, m_1, m_1), (4, m_2, m_2), \mathsf{eq})\ ((4, m_1, m_1), (4, m_2, m_2), \mathsf{eq})$

  - via proact–$\mathsf{imove}_{1,2}$ (more precisely, by choosing the proact disjunct in Def. 8 while taking w to be 0, and transitioning with both vstates synchronously, i.e., choosing the $\mathsf{imove}_{1,2}$ disjunct in Fig. 6's definition of proact).
- Finally, we move:
  - from $\Delta\, 0\, (\infty, \infty)\, ((4, m_1, m_1), (4, m_2, m_2), st)\, ((4, m_1, m_1), (4, m_2, m_2), st)$
  - to $\Delta\, \infty\, (\infty, \infty)\, (\bot, \bot, st)\, (\bot, \bot, st)$ where st is eq if $m_1 = m_2$ and diff if $m_1 \neq m_2$
  - via react–$\mathsf{match}^{1,2}$–$\mathsf{match}^{1,2}_{1,2}$.

  Note that this last move can (synchronously) change the observation divergence statuses from eq to diff, because this is the moment when the observations may differ, namely the observations m_1 and m_2 respectively.

The above argument therefore shows that our chosen Δ is an unwinding. □

This unwinding proof ensures relative security given our soundness theorem (forthcoming in §5.5). Much of the intuition for soundness relies on the fact that the state manipulation from the unwinding conditions captures the growth of the counterpart traces from the statement of relative security. On the other hand, as already pointed out, the role of the numeric timers v, v_1, v_2 is essential to ensure the fairness of this simulation process, affecting its ability to properly capture this growth. Notice how, in the unwinding proof sketch above, we started with an arbitrary $v \in \mathbb{N} \cup \{\infty\}$ which:

- became ∞ after taking a reaction move;
- then had to be decreased (and to make a choice we made it 0) when taking a proactive move;
- became ∞ again upon the next reaction move.

This design ensures that proactive moves are (eventually) alternated with reacting moves, thus avoiding "filibustering" with proactive moves, which would destroy soundness. To see what would happen without this timer guardrail, let us modify our example slightly.

Example 11 We modify Example 10 by changing the transition $(3, m, 0) \Rightarrow (4, m, m)$ to $(3, m, 0) \Rightarrow (4, m, 0)$, which has the effect of removing the leak from the vanilla system (while the leak still remains in the optimization-enhanced system, via $(2, m, 0) \Rightarrow (4, m, m)$); consequently, relative security now fails. And indeed, the unwinding proof from Example 10 no longer works, because now, when in $\Delta\, \infty\, (\infty, \infty)\, ((4, m_1, m_1), (4, m_2, m_2), \mathsf{eq})\, ((3, m_1, 0), (3, m_2, 0), \mathsf{eq})$ we must move to $\Delta\, 0\, (\infty, \infty)\, ((4, m_1, m_1), (4, m_2, m_2), \mathsf{eq})\, ((4, m_1, 0), (4, m_2, 0), \mathsf{eq})$. From here, the final reacting move to $\Delta\, \infty\, (\infty, \infty)\, (\bot, \bot, st)\, (\bot, \bot, st)$ with st = (if $m_1 = m_2$ then eq else diff) would be impossible—because, when $m_1 \neq m_2$, the ostates diverge observationally but the vstates cannot, as stipulated in the definition of $\mathsf{match}^{0,1}_{0,1}$.

However, an alternative is to move:

- from $\Delta\, \infty\, (\infty, \infty)\, ((4, m_1, m_1), (4, m_2, m_2), \mathsf{eq})\, ((3, m_1, 0), (3, m_2, 0), \mathsf{eq})$
- to $\Delta\, w\, (\infty, \infty)\, ((4, m_1, m_1), (4, m_2, m_2), \mathsf{eq})\, ((3, m_1, 0), (3, m_2, 0), \mathsf{eq})$ for some $w < \infty$
- via proact–$\mathsf{imove}_{1,2}$.

This is possible thanks to the (self) vtransitions $(3, m_1, 0) \Rightarrow (3, m_1, 0)$ and $(3, m_2, 0) \Rightarrow (3, m_2, 0)$ (and to other circumstances that make proactive moves possible, notably the lack of secret production at states with the form $(3, \_, \_)$). For the same reasons, a proactive move from $\Delta\, w\, (\infty, \infty)\, ((4, m_1, m_1), (4, m_2, m_2), \mathsf{eq})\, ((3, m_1, 0), (3, m_2, 0), \mathsf{eq})$ to $\Delta\, w'\, (\infty, \infty)\, ((4, m_1, m_1), (4, m_2, m_2), \mathsf{eq})\, ((3, m_1, 0), (3, m_2, 0), \mathsf{eq})$ for some $w' < w$, would

be possible. Should we have no guardrail from the timer argument w (which cannot decrease indefinitely due to the well-foundedness of the order on $\mathbb{N} \cup \{\infty\}$), the above argument would deem Δ an unwinding, i.e., an unsound unwinding "proof" for relative security. □

Note that, in an unwinding proof, while proactive moves decrease the timer v, reactive moves "refill" it to ∞ to allow for maximum flexibility, i.e., allow for any desired (finite) number of consecutive proactive moves in the future. This is why we allow the timers to be ∞—because it conveniently delays the choice of the exact numbers, and allows for these numbers be unbounded. In the Example 10 unwinding proof, when decreasing v from ∞, we set this number to 0 because we needed no more proactive moves (but any other finite number would have worked).

So far in our examples, the other two timer arguments, v_1, v_2, have been set to ∞ and stayed there. This is because the optimization-enhanced system had no additional infinite traces compared to the vanilla system, so there was no need to deploy these extra guardrails. However, as we explained in the "Upgrading to general unwinding" paragraph of §5.2, infinite traces can bring further opportunities for unsoundness by filibustering, which these additional timers prevent. The next example illustrates this.

Example 12 Consider the following system and attacker models:

- $\mathcal{SM}_{\mathsf{van}}$: $\mathsf{State}_{\mathsf{van}} = \{1, 2, 3, 4, 5, 6\}$; $\mathsf{istate}_{\mathsf{van}}(s)$ iff $s \in 1, 2$; and vtransitions $1 \Rightarrow 3 \Rightarrow 3$ and $2 \Rightarrow 4 \Rightarrow 4$.
- $\mathcal{SM}_{\mathsf{opt}}$: Same as $\mathcal{SM}_{\mathsf{van}}$ with additional transitions $1 \Rightarrow 5 \Rightarrow 5$ and $2 \Rightarrow 6 \Rightarrow 6$
- $\mathcal{AM}_{\mathsf{van}}$ has $\mathsf{Sec}_{\mathsf{van}} = \mathsf{Obs}_{\mathsf{van}} = \mathsf{State}_{\mathsf{van}}$, $\mathsf{Act}_{\mathsf{van}} = \{\bot\}$ and
  - $\mathsf{isSec}_{\mathsf{van}}(i) \leftrightarrow i \in \{1, 2, 3, 4\}$, $\mathsf{isInt}_{\mathsf{van}}$ is vacuously true,
  - $\mathsf{getSec}_{\mathsf{van}}(i) = \mathsf{getObs}_{\mathsf{van}}(i) = i$, $\mathsf{getAct}_{\mathsf{van}} = \bot$.
- $\mathcal{AM}_{\mathsf{opt}}$ is given by the same predicates and functions as $\mathcal{AM}_{\mathsf{van}}$.

Relative security fails for these systems. The infinite otraces $\pi_1 = 1\ 5^\infty$ and $\pi_2 = 2\ 6^\infty$ have $\mathsf{A}_{\mathsf{opt}}(\pi_1) = \bot^\infty = \mathsf{A}_{\mathsf{opt}}(\pi_2)$ and $\mathsf{O}_{\mathsf{opt}}(\pi_1) = 1\ 5^\infty \neq 2\ 6^\infty = \mathsf{O}_{\mathsf{opt}}(\pi_2)$. Moreover, $\mathsf{S}_{\mathsf{opt}}(\pi_1) = 1$ and $\mathsf{S}_{\mathsf{opt}}(\pi_2) = 2$ (i.e., these otraces leak the secret sequence pair $(1, 2)$) and there clearly exists no vtrace $\hat{\pi}$ such that $\mathsf{S}_{\mathsf{van}}(\hat{\pi}) = 1$ since the only two vtraces are $\hat{\pi}_1 = 1\ 3^\infty$ and $\hat{\pi}_2 = 2\ 4^\infty$, for which we have $\mathsf{S}_{\mathsf{van}}(\hat{\pi}_1) = 1\ 3^\infty$ and $\mathsf{S}_{\mathsf{van}}(\hat{\pi}_2) = 2\ 4^\infty$. These vtraces leak the secret sequence pair $(1\ 3^\infty, 2\ 4^\infty)$, which have the secret sequences from the optimized system leak, $(1, 2)$, as prefixes.

If the v_1 and v_2 timer arguments of unwindings were ignored, then we could construct an unsound unwinding "proof". Indeed, we could take an unwinding relation Δ that would essentially match the optimization-enhanced system leak $(1, 2)$ with the vanilla system leak of $(1\ 3^\infty, 2\ 4^\infty)$, using matching moves that make progress with the vstate transitions, preventing the exploration of the remainders 3^∞ and 4^∞, thus blurring the difference between the two leaks. More precisely (removing the v_1 and v_2 timers), we would take $\Delta\ v\ (s_1, s_2, st)\ (\hat{s}_1, \hat{s}_2, \hat{st})$ to be the disjunction of the following statements:

- $s_1 = s_2 = \hat{s}_1 = \hat{s}_2$ and $st = \hat{st} = \mathsf{eq}$;
- $(s_1, s_2, \hat{s}_1, \hat{s}_2) \in \{(5, 6, 3, 4),\ (6, 5, 4, 3)\}$ and $st = \hat{st} = \mathsf{diff}$.

Then, for the critical part of the "proof" when trying to reproduce in the vanilla system the aforementioned leak that occurs in the optimization-enhanced system, we would start in $\Delta\ v\ (1, 2, \mathsf{eq})\ (1, 2, \mathsf{eq})$ and:

- move to $\Delta\ \infty\ (5, 6, \mathsf{diff})\ (3, 4, \mathsf{diff})$ via $\mathsf{react}\text{–}\mathsf{match}^{1,2}\text{–}\mathsf{match}^{1,2}_{1,2}$ (along the transitions $1 \Rightarrow 5$ and $2 \Rightarrow 6$ matched by $1 \Rightarrow 3$ and $2 \Rightarrow 4$ respectively, taking advantage of the fact that in react the only non-trivial disjunct is the $\mathsf{match}^{1,2}$ due to $\mathsf{isInt}_{\mathsf{opt}}$ holding

for 1 and 2); note that the divergence status has become diff on both sides, due to the observations in states 1 and 2 being different;

- from here, move to (i.e., stay in) $\Delta \infty$ (5, 6, diff) (3, 4, diff) via react–match^{1,2}–ignore, where "ignore" indicates the first disjunct in the definition of match^{1,2} (along the otransitions $5 \Rightarrow 5$ and $6 \Rightarrow 6$, matched by an idle move and 3 or 4, which is allowed because isSec does not hold for 5 or 6).

However, the timer arguments v_1 and v_2 prevent us from performing the above unwinding "proof", because they are required to decrease when we take a react–match^{1,2}–ignore move. This would make it impossible to indefinitely stay in $\Delta \infty$ (5, 6, diff) (3, 4, diff), because the moves will instead be $\Delta \infty (v_1, v_2)$ (5, 6, diff) (3, 4, diff) to $\Delta \infty (v_1', v_2')$ (5, 6, diff) (3, 4, diff) to $\Delta \infty (v_1'', v_2'')$ (5, 6, diff) (3, 4, diff) etc. with $(v_1, v_2) > (v_1', v_2') > (v_1'', v_2'') > \ldots$. Thus, the conditions on v_1, v_2 ensure the ostates co-evolve with the vstates in the unwinding, reflecting that in the definition of relative security, the vtraces must grow as their counterpart otraces grow. □

5.4 Formalizing the Unwinding Definition

The formalizations for the unwinding definitions presented in (§5.2) can be found in the *Unwinding* and *Unwinding_fin* theories, both building directly on the relative security theory from (§4) as shown in Fig. 2.

In general, the formalization aims to keep the same notation and syntax as in (§5.2). Both theories begin by setting up the definitions presented in this section within the *Rel_Sec* locale context. We define each part of the proact(Δ) and react(Δ) predicates using Isabelle definitions. For example, the Isabelle definition *match1_1* is equivalent to match$^1_1(\Delta)$ in Fig. 5.

```
definition "match1_1 Δ v1 v2 s1 s1' s2 statA sv1 sv2 statO ≡
    ∃sv1'. validTransV (sv1,sv1') ∧
       Δ ∞ v1 v2 s1' s2 statA sv1' sv2 statO"
```

Additionally, numerous helper lemmas are proven on basic properties (such as the monotonicity of the unwinding relation with respect to match predicates). To give a feel for the similarities in our formal notation, we show *unwindCond* in full (the formalization of Def. 8):

```
definition unwindCond :: "(enat ⇒ enat ⇒ enat ⇒ 'stateO ⇒ 'stateO
  ⇒ status ⇒ 'stateV ⇒ 'stateV ⇒ status ⇒ bool) ⇒ bool"  where
  "unwindCond Δ ≡ ∀v v1 v2 s1 s2 statA sv1 sv2 statO.
   reachO s1 ∧ reachO s2 ∧ reachV sv1 ∧ reachV sv2 ∧
   Δ v v1 v2 s1 s2 statA sv1 sv2 statO     ⟶
   (finalO s1 ⟷ finalO s2) ∧ (finalV sv1 ⟷ finalO s1)
   ∧ (finalV sv2 ⟷ finalO s2)  ∧ (statA = Eq ⟶ (isIntO s1 ⟷ isIntO
s2))
   ∧  ((∃w < v. proact Δ w v1 v2 s1 s2 statA sv1 sv2 statO)
       ∨  react Δ v1 v2 s1 s2 statA sv1 sv2 statO )"
```

Both Unwinding theories contain the same definitions (and supporting lemmas), however the *Unwinding_fin* theory definitions omit the additional timer variables (*v1* and *v2* above), thus "ignoring everything highlighted in gray" from Fig. 5 and Fig. 6.

5.5 Soundness of the Unwinding Proof Method

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

Thm 13 (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 $s_1, s_2 \in \mathsf{State}_{\mathsf{opt}}$ 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. $(\mathcal{SM}_{\mathsf{opt}}, \mathcal{AM}_{\mathsf{opt}}) \geq^{\mathsf{fin}}_{\checkmark} (\mathcal{SM}_{\mathsf{van}}, \mathcal{AM}_{\mathsf{van}})$).

In Isabelle, this corresponds to the `unwind_lrsecure` theorem statement in the `Relative_Security` theory (or `unwind_rsecure` for the finitary version in `Relative_Security_fin`). The Isabelle statements use the `initCond` definition to encapsulate the second assumption and encourage modularity in proofs.

```
theorem unwind_lrsecure:
  assumes init: "initCond Δ" and unwind: "unwindCond Δ"
   shows lrsecure
```

The proof of this theorem, while formally laborious, essentially follows the ideas we discussed in §5.1 and §5.2, where we explained what the unwinding conditions are meant to achieve. A detailed proof sketch with links to the formalization is provided at the end of this section (§5.8).

The relative security proof method. Thm. 13 enables a sound proof method to ensure relative security of a program, i.e. it suffices to provide an unwinding relation that covers the initial state.

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.

5.6 Compositional, Distributed Unwinding

In practice, to enable compositional proofs on concrete programs, a network of relations that unwind *into each other* is often required, rather than a single unwinding relation which "unwinds into itself":

Def 14 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
- $\overline{\Delta} = (\Delta_i)_{i \in \{0,\ldots,n\}}$ where $\Delta_i : \mathbb{N}_\infty \to (\mathsf{State}_{\mathsf{opt}} \times \mathsf{State}_{\mathsf{opt}} \times \mathsf{Status}) \to (\mathsf{State}_{\mathsf{van}} \times \mathsf{State}_{\mathsf{van}} \times \mathsf{Status}) \to \mathsf{Bool}$ for each i

is an *unwinding network* if 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. 8 with Δ replaced with $\bigvee_{j \in \mathsf{next}(i)} \Delta_j$ hold.

This definition has been formalized in Isabelle as `unwindIntoCond`. It generalizes Def. 8 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 the set of the initial nodes in this network.

As already discussed, often an unwinding relation is a union (disjunction) of several smaller relations which represent certain patterns of configurations, and an unwinding proof travels between them. These smaller relations can be organized into an unwinding network.

Example 15 The unwinding relation Δ from Example 10 had the property that $\Delta\ v\ (v_1, v_2)\ (s_1, s_2, st)\ (\hat{s}_1, \hat{s}_2, \hat{st}) \leftrightarrow \bigvee_{i=1}^{4} \Delta_i\ v\ (v_1, v_2)\ (s_1, s_2, st)\ (\hat{s}_1, \hat{s}_2, \hat{st})$, where:

- $\Delta_1\ v\ (v_1, v_2)\ (s_1, s_2, st)\ (\hat{s}_1, \hat{s}_2, \hat{st})$ says $v_1 = v_2 = \infty$ and $s_1 = s_2 = \hat{s}_1 = \hat{s}_2 \in \{(1, m, 0),\ (3, m, 0),\ (4, m, m)\}$ and $st = \hat{st} = \mathsf{eq}$;
- $\Delta_2\ v\ (v_1, v_2)\ (s_1, s_2, st)\ (\hat{s}_1, \hat{s}_2, \hat{st})$ says $v_1 = v_2 = \infty$ and $s_1 = s_2 = (2, m, 0)$, $\hat{s}_1 = \hat{s}_2 = (1, m, 0)$ and $st = \hat{st} = \mathsf{eq}$;
- $\Delta_3\ v\ (v_1, v_2)\ (s_1, s_2, st)\ (\hat{s}_1, \hat{s}_2, \hat{st})$ says $v_1 = v_2 = \infty$ and $s_1 = s_2 = (4, m, m)$, $\hat{s}_1 = \hat{s}_2 = (3, m, 0)$ and $st = \hat{st} = \mathsf{eq}$;
- $\Delta_4\ v\ (v_1, v_2)\ (s_1, s_2, st)\ (\hat{s}_1, \hat{s}_2, \hat{st})$ says $v_1 = v_2 = \infty$ and $s_1 = s_2 = \hat{s}_1 = \hat{s}_2 = \bot$ and $st = \hat{st}$.

Indeed, the proof that Δ is an unwinding (as sketched in Example 10) is essentially a more refined proof that $(4, \mathsf{next}, \mathsf{Init}, (\Delta_i)_{i\in\{1,2,3,4\}})$ is an unwinding network, where:

- $\mathsf{next}(1) = \{1, 4\}$, $\mathsf{next}(2) = \{3\}$, $\mathsf{next}(3) = \{2\}$, $\mathsf{next}(4) = \emptyset$.
- $\mathsf{Init} = \{1, 2\}$. □

Thm 16 (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. 13). Then $(\mathcal{SM}_{\mathsf{opt}}, \mathcal{AM}_{\mathsf{opt}}) \geq_{\checkmark} (\mathcal{SM}_{\mathsf{van}}, \mathcal{AM}_{\mathsf{van}})$.

Note that a similar generalization holds for finitary unwinding.

Proof sketch We can verify that the union of all the relations in the network, $\bigcup_{i=1}^{n} \Delta_i$, satisfies the hypotheses of Thm. 13. □

This has been formalized in the `distrib_unwind_(l)rsecure` theorem in Isabelle. Unlike the formal proof of Thm. 13, the formal proof is almost identical for both the finitary and general case. Following the proof sketch above, the majority of the formal proof comes in establishing the hypothesis—in particular establishing the unwinding condition—after which the resulting proof goals can mostly be discharged using automated tactics.

Thm. 16 is useful in our verification case studies (to be described in §8), where different unwinding components turned out to naturally correspond to the different phases that the considered programs' executions go through.

5.7 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). To handle 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 $(\hat{\pi}_1, \hat{\pi}_2)$

$$(s,\sigma) \Rightarrow_{\mathsf{isSec}}^{\mathsf{getSec}} (s',\sigma') \doteq s \Rightarrow s' \wedge ((\neg \mathsf{isSec}(s) \wedge \sigma = \sigma') \vee (\mathsf{isSec}(s) \wedge \sigma = \mathsf{getSec}(s) \cdot \sigma'))$$

$$\mathsf{move}_1(\Gamma)\ (s_1,\sigma_1)\ (s_2,\sigma_2) \doteq \forall s_1', \sigma_1'.\ (s_1,\sigma_1) \Rightarrow_{\mathsf{isSec}}^{\mathsf{getSec}} (s_1',\sigma_1') \longrightarrow \Gamma\ (s_1',\sigma_1')\ (s_2,\sigma_2)$$

$$\mathsf{move}_2(\Gamma)\ (s_1,\sigma_1)\ (s_2,\sigma_2) \doteq \forall s_2', \sigma_2'.\ (s_2,\sigma_2) \Rightarrow_{\mathsf{isSec}}^{\mathsf{getSec}} (s_2',\sigma_2') \longrightarrow \Gamma\ (s_1,\sigma_1)\ (s_2',\sigma_2')$$

$$\mathsf{move}_{1,2}(\Gamma)\ (s_1,\sigma_1)\ (s_2,\sigma_2) \doteq \forall s_1', \sigma_1', s_2', \sigma_2'.\ (s_1,\sigma_1) \Rightarrow_{\mathsf{isSec}}^{\mathsf{getSec}} (s_1',\sigma_1') \wedge (s_2,\sigma_2) \Rightarrow_{\mathsf{isSec}}^{\mathsf{getSec}} (s_2',\sigma_2') \longrightarrow \Gamma\ (s_1',\sigma_1')\ (s_2',\sigma_2')$$

Fig. 7 The defining predicates for SD unwinding.

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 develop unwinding conditions ensuring that no vtraces starting in s_1 and s_2 can execute the same actions and make 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 17 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 the following hold (where we omit the van subscript, thus writing isInt instead of $\mathsf{isInt}_{\mathsf{van}}$ etc.):

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

The predicates $\mathsf{move}_1(\Gamma)$, $\mathsf{move}_2(\Gamma)$ and $\mathsf{move}_{1,2}(\Gamma)$ are defined in Fig. 7 (where we again omit the van subscript). They are based on the secret-directed transition relation $\Rightarrow_{\mathsf{isSec}}^{\mathsf{getSec}}$, also defined in Fig. 7.

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 of equal actions. Additionally, the secret-directed transition relation $\Rightarrow_{\mathsf{isSec}}^{\mathsf{getSec}}$ shown in Fig. 7 ensures 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. 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 18 (The Disproof Unwinding Theorem) Assuming:

- $\pi_1, \pi_2 \in \mathsf{Trace}_{\mathsf{opt}}$, $\mathsf{A}_{\mathsf{opt}}(\pi_1) = \mathsf{A}_{\mathsf{opt}}(\pi_2)$ and $\mathsf{O}_{\mathsf{opt}}(\pi_1) \neq \mathsf{O}_{\mathsf{opt}}(\pi_2)$.
- Γ is an SD unwinding.
- Γ covers the initial states w.r.t. $(\mathsf{S}_{\mathsf{opt}}(\pi_1), \mathsf{S}_{\mathsf{opt}}(\pi_2))$, 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., $(\mathcal{SM}_{\mathsf{opt}}, \mathcal{AM}_{\mathsf{opt}}) \not\preceq_{\checkmark} (\mathcal{AM}_{\mathsf{van}}, \mathcal{LM}_{\mathsf{van}})$.

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

Proof sketch The following more general fact follows by coinduction on sequence (lazy-list) equality: if Γ is an SD unwinding, then, for all $s_1, \sigma_1, s_2, \sigma_2, \hat{\pi}_1, \hat{\pi}_2$ such that $\Gamma\ (s_1,\sigma_1)\ (s_2,\sigma_2)$, $\hat{\pi}_1 \in \mathsf{Trace}_{\mathsf{van}}$, $\hat{\pi}_2 \in \mathsf{Trace}_{\mathsf{van}}$, $\hat{\pi}_1$ starts in s_1, $\hat{\pi}_2$ starts in s_2, $\mathsf{S}\ \hat{\pi}_1 = \sigma_1$, $\mathsf{S}\ \hat{\pi}_2 = \sigma_2$ and $\mathsf{A}\ \hat{\pi}_1 = \mathsf{A}\ \hat{\pi}_2$, we have that $\mathsf{O}\ \hat{\pi}_1 = \mathsf{O}\ \hat{\pi}_2$. □

Example 19 Consider the following system models and state-wise attacker models:

- $\mathcal{SM}_{\mathsf{van}}$: $\mathsf{State}_{\mathsf{van}} = (\mathbb{N}^3)_\perp$; $\mathsf{istate}_{\mathsf{van}}((i,m,n))$ iff $i = 1 \wedge n = 0$; and vtransitions $(1,m,0) \Rightarrow (2,m,0) \Rightarrow \perp$ if m is even, and $(1,m,0) \Rightarrow (2,m,m) \Rightarrow \perp$ if m is odd. Hence $\mathsf{Trace}_{\mathsf{van}}$ consists of traces $(1,m,0)\,(2,m,0)\,\perp$ for some even m, or $(1,m,0)\,(2,m,m)\,\perp$ for some odd m.
- $\mathcal{SM}_{\mathsf{opt}}$: $\mathsf{State}_{\mathsf{opt}} = \mathsf{State}_{\mathsf{van}}$; $\mathsf{istate}_{\mathsf{opt}}((i,m,n))$ iff $\mathsf{istate}_{\mathsf{van}}((i,m,n))$ or $i = 3 \wedge n = 0$; and otransitions extend the vtransitions to include $(3,m,0) \Rightarrow (4,m,m) \Rightarrow \perp$ for all m,n. Thus, $\mathsf{Trace}_{\mathsf{opt}}$ consists of all traces in Trace_van and those of the form $(3,m,0)\,(4,m,m)\,\perp$.
- $\mathcal{AM}_{\mathsf{van}}$ has $\mathsf{Sec} = \mathsf{Act} = \mathsf{Obs} = \mathbb{N}$, with the following predicates and functions:
  - $\neg\, \mathsf{isSec}_{\mathsf{van}}(\perp)$ and $\mathsf{isSec}_{\mathsf{van}}(i,m,n) \iff i \in \{1,3\}$;
  - $\mathsf{isInt}_{\mathsf{van}}$ is true in all states except for $\perp$;
  - $\mathsf{getSec}(i,m,n) = m$, $\mathsf{getAct}(i,m,n) = i$ and $\mathsf{getObs}(i,m,n) = n$.
- $\mathcal{AM}_{\mathsf{opt}}$ is given by the same predicates and functions as $\mathcal{AM}_{\mathsf{van}}$.

Both relative security and finitary relative security fail for these systems. Indeed, considering the otraces $\pi_1 = (3,0,0)\,(4,0,0)\,\perp$ and $\pi_2 = (3,2,0)\,(4,2,2)\,\perp$, we have that $\mathsf{A}_{\mathsf{opt}}(\pi_1) = 3\,4 = \mathsf{A}_{\mathsf{opt}}(\pi_2)$ and $\mathsf{O}_{\mathsf{opt}}(\pi_1) = 0\,0 \neq 0\,2 = \mathsf{O}_{\mathsf{opt}}(\pi_2)$. Moreover, we have that $\mathsf{S}_{\mathsf{opt}}(\pi_1) = 0$ and $\mathsf{S}_{\mathsf{opt}}(\pi_2) = 2$, i.e., the secret from second component of the initial state leaks through the observation made in the third component of the final state. On the other hand, the only vtraces $\hat{\pi}_1$ and $\hat{\pi}_2$ such that $\mathsf{S}_{\mathsf{van}}(\hat{\pi}_1) = 0$ and $\mathsf{S}_{\mathsf{van}}(\hat{\pi}_2) = 2$ are $\hat{\pi}_1 = (1,0,0)\,(2,0,0)\,\perp$ and $\hat{\pi}_2 = (1,2,0)\,(2,2,0)\,\perp$. But for these, we have $\mathsf{O}_{\mathsf{van}}(\hat{\pi}_1) = 0\,0 = \mathsf{O}_{\mathsf{van}}(\hat{\pi}_2)$, so they produce no leak.

The above sketched failure of relative security can be shown by SD unwinding as follows. We choose the above traces π_1 and π_2 (with the secret sequences 0 and 2), and define $\Gamma\,(s_1,\sigma_1)\,(s_2,\sigma_2)$ to be the disjunction of the following statements:

- $s_1 = (1,m_1,0)$ and $s_2 = (1,m_2,0)$ for some m_1, m_2, $\sigma_1 = 0$ and $\sigma_2 = 2$;
- $s_1 = (2,0,0)$, $s_2 = (2,2,0)$ and $\sigma_1 = \sigma_2 = \epsilon$ (the empty sequence);
- $s_1 = s_2 = \perp$ and $\sigma_1 = \sigma_2 = \epsilon$.

For the SD unwinding proof, we take the first disjunct and start with $\Gamma\,(s_1, \mathsf{S}_{\mathsf{opt}}(\pi_1))\,(s_2, \mathsf{S}_{\mathsf{opt}}(\pi_2))$ where s_1 and s_2 are initial, i.e., with $\Gamma\,((1,m_1,0),0)\,((1,m_2,0),2)$ for some m_1, m_2. Of the conditions from Def. 17, the first two hold trivially because isInt is true for $(1,m_1,0)$ and $(1,m_2,0)$. For the third, we need to verify:

- $\mathsf{getObs}(1,m_1,0) = \mathsf{getObs}(1,m_2,0)$, which is true because both are 0;
- $\mathsf{move}_{1,2}(\Gamma)\,((1,m_1,0),0)\,((1,m_2,0),2)$.

Expanding Fig. 7's definition of $\mathsf{move}_{1,2}$, let (s_1',σ_1') and (s_2',σ_2') be such that

- $((1,m_1,0),0) \Rightarrow^{\mathsf{getSec}}_{\mathsf{isSec}} (s_1',\sigma_1')$ and
- $((1,m_2,0),2) \Rightarrow^{\mathsf{getSec}}_{\mathsf{isSec}} (s_2',\sigma_2')$.

Then, since $\mathsf{isSec}(1,m_1,0)$ and $\mathsf{isSec}(1,m_2,0)$ hold and $\mathsf{getSec}(1,m_1,0) = m_1$ and $\mathsf{getSec}(1,m_2,0) = m_2$, the only possibility is that $m_1 = 0$, $m_2 = 2$, $s_1' = (2,0,0)$, $s_2' = (2,2,0)$, and $\sigma_1' = \sigma_2' = \epsilon$. So this brings us to $\Gamma\,((2,0,0),\epsilon)\,((2,2,0),\epsilon)$.

Now assume $\Gamma\,((2,0,0),\epsilon)\,((2,2,0),\epsilon)$ (i.e., the second disjunct in the definition of Γ). Of the conditions from Def. 17, the first two again hold trivially because isInt is true for $(2,0,0)$ and $(2,2,0)$. For the third, we need to verify:

- $\mathsf{getObs}((2,0,0)) = \mathsf{getObs}((2,2,0))$, which is true because both are 0;
- $\mathsf{move}_{1,2}(\Gamma)\,((2,0,0),\epsilon)\,((2,2,0),\epsilon)$.

Expanding the definition of $\mathsf{move}_{1,2}$, let (s_1',σ_1') and (s_2',σ_2') be such that

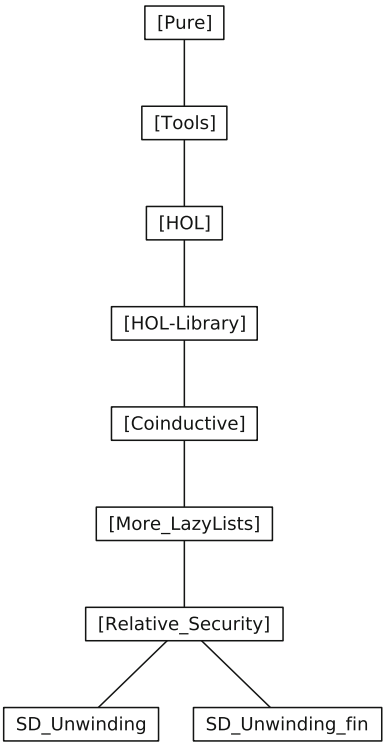


Fig. 8 Theory structure of the SD Unwinding Isabelle mechanization.

- $((2,0,0),\epsilon) \Rightarrow^{\mathsf{getSec}}_{\mathsf{isSec}} (s_1', \sigma_1')$ and
- $((2,2,0),\epsilon) \Rightarrow^{\mathsf{getSec}}_{\mathsf{isSec}} (s_2', \sigma_2')$.

Then, since $\neg\ \mathsf{isSec}(2,0,0)$ and $\neg\ \mathsf{isSec}(2,2,0)$ hold, the only possibility is that $s_1' = s_2' = \bot$, and $\sigma_1' = \sigma_2' = \epsilon$. So this brings us to $\Gamma\ (\bot,\epsilon)\ (\bot,\epsilon)$.

Finally, assume $\Gamma\ (\bot,\epsilon)\ (\bot,\epsilon)$ (the third disjunct of Γ). All the conditions from Def. 17 are trivially true, because isInt does not hold for $\bot$, and $\bot$ is final which means $\mathsf{move}_1(\Gamma)\ (\bot,\epsilon)\ (\bot,\epsilon)$ and $\mathsf{move}_2(\Gamma)\ (\bot,\epsilon)\ (\bot,\epsilon)$ are trivially true. □

The formalization work for SD unwinding is located in a separate Isabelle AFP entry [7]. The session graph generated for this work is shown in Fig. 8. Note that the [Relative_Security] box encompasses the entire relative security Isabelle session shown in Fig. 2 (which is imported as the SD theories), build on the same `Relative_Security` theory.

The formalization follows the same approach as unwinding, with Def. 17 being represented in Isabelle by `lunwindSDCond` (or `unwindSDCond` in the finitary case). The formal definitions are practically identical except for the use of lazy lists in the general case.

Similarly, Thm. 18 is formalized as `unwindSD_lrsecure`. Each of the labelled assumptions `tr14`, `init`, and `unw` line up with the hypothesis of the theorem statement in the formal statement below.

```
theorem unwindSD_lrsecure:
  assumes tr14: "istateO s1" "Opt.lvalidFromS s1 tr1" "lcompletedFromO s1 tr1"
  "istateO s2" "Opt.lvalidFromS s2 tr2" "lcompletedFromO s2 tr2"
  "Opt.lA tr1 = Opt.lA tr2" "Opt.lO tr1 ≠ Opt.lO tr2"
  and init: "⋀sv1 sv2. istateV sv1 ⟹ corrState sv1 s1
```

```
⟹ istateV sv2 ⟹ corrState sv2 s2 ⟹
    Γ sv1 (Opt.lS tr1) sv2 (Opt.lS tr2)"
  and unw: "lunwindSDCond Γ"
  shows "¬ lrsecure"
```

The formal proof of this statement is far more straightforward than what was required for Thm. 13, however once again the general case is noticeably more involved than the finitary case. In the finitary case, we prove a single proposition, `unwindSDCond_aux` which is the formal statement of the property in the proof sketch above for finite traces. The proof proceeds by normal induction on the length of the traces (i.e., coinduction is unnecessary), and a case split on if `isIntV sv1` is true, i.e. the interaction predicate on a vstate.

Comparatively, the same simple inductive approach in the general case is only used for a lemma reasoning on the unwinding relation as applied to last states and finite traces where interaction never occurs (`unwindSDCond_aux_inductive`). This is then used to establish a slightly more general lemma `unwindSDCond_inductive` which can be used to prove that the hypothesis of the property holds (again for traces with no interaction). The proposition `lunwindSDCond_aux` then formalizes the property using coinduction, as outlined in the proof sketch, with the helper lemmas being used to establish the equality of the first elements of the observation sequences.

5.8 Formally Verifying Soundness

The soundness proof for Thm. 13 is the most involved part of the formalization, particularly for the infinite-trace case. The formalization revealed the need for several substantial helper lemmas beyond the intuitive ideas gained from defining the unwinding relation presented in (§5.1). The infinite-trace case was substantially more involved than the finite-trace one, reflected in the final 7.5K LOC required for the `Unwinding` in comparison to 1.2K LOC for the `Unwinding_fin` theory. Infinitary unwinding was also the driving factor behind the lazy list extensions, including some further extensions in `Trivia` (0.6K LOC) providing custom support for the inductive-coinductive mixture of reasoning called for by the timers. We refer the interested reader to the formalization for the full formal proof [6]. Here we sketch the proof in natural language with links to the formalization.

Reasoning on trace fragments. To prove Thm. 13, we need a stronger property involving fragments of traces. A *(possibly infinite) trace fragment* starting in s is a non-empty sequence of states $s_0\, s_1 \ldots$ such that $s_0 = s$ and $s_i \rightarrow s_{i+1}$ for all i smaller than the (possibly infinite) length of the sequence. We write Frag_s for the set of trace fragments starting in s and $\mathsf{Frag}_{s,s'}$ for the set of finite trace fragments starting in s and ending in s'.

Lemma 20 establishes a property on finite fragments which is stronger than relative security in some respects and weaker in others, as we will explain below.

Lemma 20 Assume Δ is an unwinding and $v, v_1, v_2 \in \mathbb{N}_\infty, s_1, s_2, s_1', s_2' \in \mathsf{State}_{\mathsf{opt}}, \hat{s}_1, \hat{s}_2 \in \mathsf{State}_{\mathsf{van}}, st, \hat{st} \in \mathsf{Status}$,
$\pi_1 \in \mathsf{Frag}^{\mathsf{opt}}_{s_1,s_1'}$ and $\pi_2 \in \mathsf{Frag}^{\mathsf{opt}}_{s_2,s_2'}$ such that:

- $\Delta\ v\ (v_1, v_2)\ (s_1, s_2, st)\ (\hat{s}_1, \hat{s}_2, \hat{st})$ holds.
- $\mathsf{A}(\pi_1) = \mathsf{A}(\pi_2)$.
- $\mathsf{isInt}(s_1')$ and $\mathsf{isInt}(s_2')$ hold.

Then there exist $\hat{s}_1', \hat{s}_2' \in \mathsf{State}_{\mathsf{van}}$, $\hat{\pi}_1 \in \mathsf{Frag}^{\mathsf{van}}_{\hat{s}_1,\hat{s}_1'}$, $\hat{\pi}_2 \in \mathsf{Frag}^{\mathsf{van}}_{\hat{s}_2,\hat{s}_2'}$ and $\hat{st}, \hat{st}' \in \mathsf{Status}$, $w_1, w_2 \in \mathbb{N}_\infty$ such that the unwinding is sound, i.e:

- $\mathsf{S}(\hat{\pi}_1) = \mathsf{S}(\pi_1)$ and $\mathsf{S}(\hat{\pi}_2) = \mathsf{S}(\pi_2)$ and $\mathsf{A}(\hat{\pi}_1) = \mathsf{A}(\hat{\pi}_2)$
- if $st = \mathsf{eq}$ and $\mathsf{O}(\pi_1) \neq \mathsf{O}(\pi_2)$, then $\mathsf{O}(\hat{\pi}_1) \neq \mathsf{O}(\hat{\pi}_2)$ and $\hat{st}' = \mathsf{diff}$
- $\Delta \infty\ (w_1, w_2)\ (s_1', s_2', st')\ (\hat{s}_1', \hat{s}_2', \hat{st}')$ holds
- if $st' = \mathsf{eq}$ then $st = \mathsf{eq}$ and $\hat{st}' = \mathsf{eq}$

Proof sketch The proof is by lexicographic induction on $|\pi_1| + |\pi_2|$ (the sum of lengths of the otrace fragments) as first criterion and v as second criterion. Each reactive step decreases $|\pi_1| + |\pi_2|$ and each proactive step decreases v. The conditions from the definition of unwinding ensure that the relationships stated in the theorem between the four finite trace fragments, $\pi_1, \pi_2, \hat{\pi}_1, \hat{\pi}_2$, are preserved by the induction step. In the conclusion of the lemma, we have ∞ as the first timer argument for Δ; this is because in the base case, where π_1 and π_2 are singletons consisting of s_1' and s_2', we take a reactive rather than a proactive move. □

Note that, on the one hand, the property stated in this lemma is *stronger* than the conditions that make up relative security, in that we require that $\hat{\pi}_1$ and $\hat{\pi}_2$ exist and have the same actions ($\mathsf{A}(\hat{\pi}_1) = \mathsf{A}(\hat{\pi}_2)$) for any π_1 and π_2 that have the same actions ($\mathsf{A}(\pi_1) = \mathsf{A}(\pi_2)$) *regardless of whether they have different observations*; and then, in case they have different observations, the counterparts also have different observations. This stronger property emerges from the incremental nature of the unwinding. Even though the observations π_1 and π_2 will diverge eventually, when building their counterparts incrementally we must also cater for the case when π_1 and π_2 have not diverged yet (a case not covered by the definition of relative security).

On the other hand, the property stated in the theorem is also *weaker* than relative security, in that it only considers finite fragments of traces (whereas, in general, traces can be infinite). Moreover, the theorem requires that s_1' and s_2', the ending states of the otraces, feature interaction ($\mathsf{isInt}(s_1')$ and $\mathsf{isInt}(s_2')$).

Lemma 20 will allow us to keep growing the vtraces for as long as their counterpart otraces still have interactions on them (i.e., isInt holds on at least one of their states). To also cover the case when they run out of interaction, we need two different lemmas. The first allows one to grow the vtraces in reaction to a transition by the left otrace:

Lemma 21 Assume Δ is an unwinding and $v, v_1, v_2 \in \mathbb{N}_\infty$, $s_1, s_2, s_1' \in \mathsf{State}_{\mathsf{opt}}$, $\hat{s}_1, \hat{s}_2 \in \mathsf{State}_{\mathsf{van}}$, $st, \hat{st} \in \mathsf{Status}$ such that:

- $\Delta\ v\ (v_1, v_2)\ (s_1, s_2, st)\ (\hat{s}_1, \hat{s}_2, \hat{st})$ holds
- $s_1 \Rightarrow s_1'$
- $\neg\, \mathsf{isInt}\ s_1$ and $\neg\, \mathsf{isInt}\ s_2$

Then there exist $\hat{s}_1', \hat{s}_2' \in \mathsf{State}_{\mathsf{van}}$, $\hat{\pi}_1 \in \mathsf{Frag}^{\mathsf{van}}_{\hat{s}_1, \hat{s}_1'}$, $\hat{\pi}_2 \in \mathsf{Frag}^{\mathsf{van}}_{\hat{s}_2, \hat{s}_2'}$ and $\hat{st}, \hat{st}' \in \mathsf{Status}$, $w_1, w_2 \in \mathbb{N}_\infty$ such that:

- $\mathsf{S}(\hat{\pi}_1) = \mathsf{S}(s_1 s_1')$ and $\mathsf{S}(\hat{\pi}_2) = []$ (i.e., isSec is false for all states in $\hat{\pi}_2$)
- $\mathsf{A}(\hat{\pi}_1) = \mathsf{A}(\hat{\pi}_2)$
- $\Delta \infty\ (w_1, w_2)\ (s_1', s_2', st')\ (\hat{s}_1', \hat{s}_2', \hat{st}')$ holds
- $\mathsf{len}(\hat{\pi}_1) \geq 2 \vee w_1 < v_1$ and $\mathsf{len}(\hat{\pi}_2) \geq 2 \vee w_2 < v_2$

The second is a symmetric lemma for the right otrace:

Lemma 22 Assume Δ is an unwinding and $v, v_1, v_2 \in \mathbb{N}_\infty$, $s_1, s_2, s_2' \in \mathsf{State}_{\mathsf{opt}}$, $\hat{s}_1, \hat{s}_2 \in \mathsf{State}_{\mathsf{van}}$, $st, \hat{st} \in \mathsf{Status}$ such that:

- $\Delta\ v\ (v_1, v_2)\ (s_1, s_2, st)\ (\hat{s}_1, \hat{s}_2, \hat{st})$ holds
- $s_2 \Rightarrow s_2'$

- $\neg$ isInt s_1 and $\neg$ isInt s_2

Then there exist $\hat{s}_1', \hat{s}_2' \in \mathsf{State}_{\mathsf{van}}$, $\hat{\pi}_1 \in \mathsf{Frag}^{\mathsf{van}}_{\hat{s}_1,\hat{s}_1'}$, $\hat{\pi}_2 \in \mathsf{Frag}^{\mathsf{van}}_{\hat{s}_2,\hat{s}_2'}$ and $\hat{st}, \hat{st}' \in \mathsf{Status}$, $w_1, w_2 \in \mathbb{N}_\infty$ such that:

- $\mathsf{S}(\hat{\pi}_1) = []$ (i.e., isSec is false for all states in $\hat{\pi}_1$) and $\mathsf{S}(\hat{\pi}_2) = \mathsf{S}(s_2 s_2')$
- $\mathsf{A}(\hat{\pi}_1) = \mathsf{A}(\hat{\pi}_2)$
- $\Delta \infty (w_1, w_2)\ (s_1', s_2', st')\ (\hat{s}_1', \hat{s}_2', \hat{st}')$ holds
- $\mathsf{len}(\hat{\pi}_1) \geq 2 \vee w_1 < v_1$ and $\mathsf{len}(\hat{\pi}_2) \geq 2 \vee w_2 < v_2$

Highlighted above is an essential property guaranteed by these two lemmas: that for each of the two otraces, either its growth is non-trivial (i.e., the fragment with which it grows has length at least 2, i.e., has at least one transition) or the corresponding timer decreases. This addresses the "filibustering" problem discussed at the end of §5.2.

The proofs of these two lemmas are by induction on w (i.e., essentially, on the maximum number of proactive moves allowed before a reactive move must occur).

Formalizing fragments. The formalization does not separately define the notion of fragment, which we introduced in the above sketch for readability. Instead, the formalization reuses the definitions and lemmas defined on traces in the core transition system locale (§4.3) such as `validTrans0` or `validFromS` which refer to valid transitions and traces (from a certain state) in the system.

Roughly, the lemmas in this section line up with the lemmas beginning with `unwindCond_ex` in the formalization, with the key lemma to the proof of Thm. 13 being defined using the Isabelle keyword **proposition** (functionally no different to **theorem** or **lemma**, but the different syntax highlights importance). Notably, the finitary case only requires a single proposition, whereas the general case has to gradually build up to this, including the development of lemmas specifically for corecursive reasoning.

Proof sketch for Thm. 13: To show that relative security holds (in the general case), let $\pi_1 \in \mathsf{Trace}_{s_1}$ and $\pi_2 \in \mathsf{Trace}_{s_2}$ such that $\mathsf{A}(\pi_1) = \mathsf{A}(\pi_2)$ and $\mathsf{O}(\pi_1) \neq \mathsf{O}(\pi_2)$.

We will first identify all states of π_1 where interaction happens (i.e., isInt holds). After that, we will use the property $\mathsf{A}(\pi_1) = \mathsf{A}(\pi_2)$ to identify corresponding points on π_2. These points form the basis of a decomposition of π_1 and π_2 into finite trace fragments that will allow the application of Lemma 20—this decomposition is depicted in Fig. 9. We distinguish two cases, depending on whether interaction happens on π_1 indefinitely or not.

Case (1): Interaction happens indefinitely, i.e., there exists an infinite number of non-empty finite otrace fragments $\rho_{1,1}, \rho_{1,2}, \ldots$ such that:

- $\pi_1 = \rho_{1,1} \cdot \rho_{1,2} \cdot \ldots$ (where $\cdot$ denotes list concatenation)
- for all $j \in \mathbb{N}$, $\mathsf{isInt}(s_{1,j})$ holds, where $s_{1,j}$ is the last state of $\rho_{1,j}$

Because $\mathsf{A}(\pi_1) = \mathsf{A}(\pi_2)$, we also have an infinite number of finite otrace fragments $\rho_{2,j}$ with their last states $s_{2,j}$ such that:

- $\pi_2 = \rho_{2,1} \cdot \rho_{2,2} \cdot \ldots$
- for all $j \in \mathbb{N}$, $\mathsf{isInt}(s_{2,j})$ holds, $\mathsf{A}(\rho_{1,j}) = \mathsf{A}(\rho_{2,j})$, and in particular $\mathsf{getAct}(s_{1,j}) = \mathsf{getAct}(s_{2,j})$

Moreover, due to $\mathsf{O}(\pi_1) \neq \mathsf{O}(\pi_2)$, there exists $j_0 \in \mathbb{N}$ such that $\mathsf{getObs}(s_{1,j_0}) \neq \mathsf{getObs}(s_{2,j_0})$, and in particular $\mathsf{O}(\rho_{1,j_0}) \neq \mathsf{O}(\rho_{2,j_0})$. In fact, choosing the smallest such j_0, we also have that $\mathsf{getObs}(s_{1,j}) = \mathsf{getObs}(s_{2,j})$ and $\mathsf{O}(\rho_{1,j}) = \mathsf{O}(\rho_{2,j})$ for all $j < j_0$.

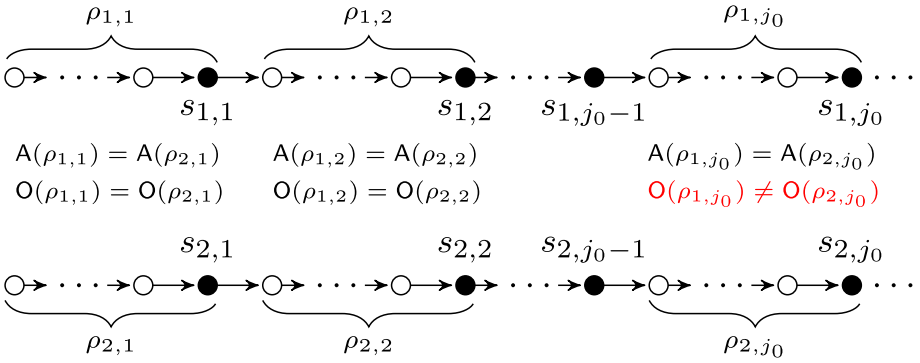


Fig. 9 Decomposing the otraces in the proof of Thm. 13. Full bullet means isInt holds, empty bullet means it doesn't.

Now, by induction on $j \in \mathbb{N}$, we define the finite vtrace fragments $\hat{\rho}_{1,j}$ and $\hat{\rho}_{2,j}$ with their last states $\hat{s}_{1,j}$ and $\hat{s}_{2,j}$ and the statuses st_j and $\hat{st}_j$ and timers $v_j^1, v_j^2 \in \mathbb{N}_\infty$ such as $\Delta \infty (v_j^1, v_j^2)\ (s_{1,j}, s_{2,j}, st_j)\ (\hat{s}_{1,j}, \hat{s}_{2,j}, \hat{st}_j)$ as follows:

- $\hat{\rho}_{1,0}, \hat{\rho}_{2,0}, st_0, \hat{st}_0, v_0^1, v_0^2$ are obtained from $\Delta \infty (v_0^1, v_0^2)\ (s_1, s_2, \mathsf{eq})\ (\hat{s}_1, \hat{s}_2, \mathsf{eq})$ using Lemma 20;
- for each j, $\hat{\rho}_{1,j+1}, \hat{\rho}_{2,j+1}, st_{j+1}, \hat{st}_{j+1}, v_{j+1}^1, v_{j+1}^2$ are obtained from $\Delta \infty (v_j^1, v_j^2)\ (s_{1,j}, s_{2,j}, st_j)\ (\hat{s}_{1,j}, \hat{s}_{2,j}, \hat{st}_j)$ using Lemma 20.

From the application of Lemma 20, we also have that, for all j, $\mathsf{A}(\hat{\rho}_{1,j}) = \mathsf{A}(\hat{\rho}_{2,j})$, $\mathsf{S}(\hat{\rho}_{1,j}) = \mathsf{S}(\rho_{1,j})$ and $\mathsf{S}(\hat{\rho}_{2,j}) = \mathsf{S}(\rho_{2,j})$. Moreover, the lemma also ensures that, for all $j < j_0$, $st_j = \hat{st}_j = \mathsf{eq}$ and $\mathsf{O}(\hat{\rho}_{1,j}) = \mathsf{O}(\hat{\rho}_{2,j})$, and then that $st_{j_0} = \hat{st}_{j_0} = \mathsf{diff}$ and $\mathsf{O}(\hat{\rho}_{1,j_0}) \neq \mathsf{O}(\hat{\rho}_{2,j_0})$.

Now, taking $\hat{\pi}_1 = \hat{\rho}_{1,1} \cdot \hat{\rho}_{1,2} \cdot \ldots$ and $\hat{\pi}_2 = \hat{\rho}_{2,1} \cdot \hat{\rho}_{2,2} \cdot \ldots$, the above properties ensure that $\mathsf{A}(\hat{\pi}_1) = \mathsf{A}(\hat{\pi}_2)$, $\mathsf{S}(\hat{\pi}_1) = \mathsf{S}(\pi_1)$ and $\mathsf{S}(\hat{\pi}_2) = \mathsf{S}(\pi_2)$, and also $\mathsf{O}(\hat{\pi}_1) \neq \mathsf{O}(\hat{\pi}_2)$, as desired.

Case (2): Interaction does not happen indefinitely, meaning there exists a *finite* number of finite otrace fragments $\rho_{1,1}, \rho_{1,2} \ldots, \rho_{1,n}$, as well as a "remainder" σ_1 such that:

- $\pi_1 = \rho_{1,1} \cdot \ldots \cdot \rho_{1,n} \cdot \sigma_1$
- for all $j \in \{1, \ldots, n\}$, $\mathsf{isInt}(s_{1,j})$ holds, where again $s_{1,j}$ is the last state of $\rho_{1,j}$

Similarly to how we proceeded in case (1), we obtain the corresponding finite otrace fragments $\rho_{2,j}$ with last states $s_{2,j}$ and the "remainder" σ_2 such that:

- $\pi_2 = \rho_{2,1} \cdot \ldots \cdot \rho_{2,n} \cdot \sigma_2$
- for all $j \in \{1, \ldots, n\}$, $\mathsf{isInt}(s_{2,j})$ holds, $\mathsf{A}(\rho_{1,j}) = \mathsf{A}(\rho_{2,j})$ and $\mathsf{getAct}(s_{1,j}) = \mathsf{getAct}(s_{2,j})$

Let us write π_1' and π_2' for the prefixes of the otraces without the remainders, i.e., $\pi_1' = \rho_{1,1} \cdot \ldots \cdot \rho_{1,n}$ and $\pi_2' = \rho_{2,1} \cdot \ldots \cdot \rho_{2,n}$. Thus, we have $\pi_1 = \pi_1 \cdot \sigma_1$ and $\pi_2 = \pi_2 \cdot \sigma_2$.

Again similarly to case (1), we can produce the (this time) finite vtraces $\hat{\pi}_1'$ and $\hat{\pi}_2'$ that satisfy the relative-security properties w.r.t. π_1' and π_2', namely $\mathsf{A}(\hat{\pi}_1') = \mathsf{A}(\hat{\pi}_2')$, $\mathsf{S}(\hat{\pi}_1') = \mathsf{S}(\pi_1')$ and $\mathsf{S}(\hat{\pi}_2') = \mathsf{S}(\pi_2')$, and also $\mathsf{O}(\hat{\pi}_1') \neq \mathsf{O}(\hat{\pi}_2')$. Moreover, we have that $\Delta \infty (v_1, v_2)\ (s_1', s_2', st)\ (\hat{s}_1', \hat{s}_2', \hat{st})$ for some timers v_1, v_2, states st and $\hat{st}$ and states $\hat{s}_1'$ and $\hat{s}_2'$, where s_1' and s_2' are the first states of σ_1 and σ_2, respectively.

What remains are the (possibly infinite) remainders σ_1 and σ_2. In other words, we are left with having to prove the following lemma:

Lemma Assume that Δ is an unwinding relation, $s_1, s_2 \in \mathsf{State}_{\mathsf{opt}}$ $\hat{s}_1, \hat{s}_2 \in \mathsf{State}_{\mathsf{van}}$, $v, v_1, v_2 \in \mathbb{N}_\infty$, $st, \hat{st} \in \mathsf{Status}$, $\sigma_1 \in \mathsf{Frag}_{s_1}$ and $\sigma_2 \in \mathsf{Frag}_{s_2}$ such that:

- $\Delta\ v\ (v_1, v_2)\ (s_1, s_2, st)\ (\hat{s}_1, \hat{s}_2, \hat{st})$
- isInt fails for all states in σ_1 and σ_2

Then there exist $\hat{\sigma}_1 \in \mathsf{Frag}_{\hat{s}_1}$ and $\hat{\sigma}_2 \in \mathsf{Frag}_{\hat{s}_1}$ such that:

- $\mathsf{A}(\hat{\sigma}_1) = \mathsf{A}(\hat{\sigma}_2)$
- $\mathsf{S}(\hat{\sigma}_1) = \mathsf{S}(\sigma_1)$ and $\mathsf{S}(\hat{\sigma}_2) = \mathsf{S}(\sigma_2)$

This lemma is proved by constructing $\hat{\sigma}_1$ and $\hat{\sigma}_2$ from σ_1 and σ_2 similarly to how we built $\hat{\pi}_1$ and $\hat{\pi}_2$ above, but instead of Lemma 20 we use Lemmas 21 and 22, alternating fairly for as long as σ_1 and σ_2 are not exhausted (since either, or both, or none, can actually be finite). □

Notes on formalizing the main proof. The formalization of the main proof [6] highlights the differences between the proof in the finitary vs general statement. In the finitary case, none of the infinite cases outlined above have to be considered. As such, the formal proof of `unwind_rsecure` in `Unwinding_fin` can apply the `unwindCond_ex` proposition directly, then use pre-existing basic lemmas on reachability of finite traces to finish the subgoals related to progress.

In comparison, the general lemma required significantly more work. We use lazy-lists to represent the possibly infinite nature of the traces, in place of the indexing in the above proof sketches. There are two main classes of lemmas in the `Unwinding` theory which were required to prove `unwind_lrsecure`. Firstly, lemmas beginning with `lvalidFrom` and `lcompletedFrom` are essentially specific *progress* lemmas, required to establish the unwindings as a result of the corecursive definitions of the two vtraces from the two otraces. Second were the lemmas that more closely mirrored the reasoning presented above, which were used to establish the (in)equalities between various secrets, actions, and observations. For example, `lS_ltrv1_ltr1` establishes the equality between the vanilla and optimized secrets. As such, many of the main formal lemmas each considered the different cases which we have separated out for readability in the proof sketch above.

Given the use of lazy-lists, many of these lemmas required the use of coinduction to prove that a property holds in the possibly infinite cases in the above proof sketch. This was the primary driver behind the extensions on the coinductive list library mentioned in (§4.2). It additionally required the setup of several coinduction rules on traces in the `Transition_System` locale.

5.9 Incompleteness of the Unwinding (Dis)Proof Methods

The question dual to soundness is completeness: Is it the case that (assuming state-wise system models), if relative security holds, then there exists an unwinding relation satisfying the hypotheses of Theorem 13? And similarly, is it the case that, if relative security fails, then there exist two otraces π_1 and π_2 and an SD unwinding relation satisfying the hypotheses of Theorem 18? The answer to both questions is negative, i.e., our unwinding (dis)proof methods are incomplete.

There are several features that make our unwinding proof method incomplete. One of them is the implicit requirement that the produced vtraces $\hat{\pi}_1$ and $\hat{\pi}_2$ *eagerly* match the secrets of their counterpart otraces π_1 and π_2 (reflecting the conditions $\mathsf{S}_{\mathsf{van}}(\hat{\pi}_1) = \mathsf{S}_{\mathsf{opt}}(\pi_1)$

and $\mathsf{S}_{\mathsf{van}}(\hat{\pi}_2) = \mathsf{S}_{\mathsf{opt}}(\pi_2)$ from the conclusion of the relative security property), regardless of whether π_1 and π_2 will end up producing an actual leak (i.e., regardless of whether $\mathsf{A}_{\mathsf{opt}}(\pi_1) = \mathsf{A}_{\mathsf{opt}}(\pi_2) \wedge \mathsf{O}_{\mathsf{opt}}(\pi_1) \neq \mathsf{O}_{\mathsf{opt}}(\pi_2)$ holds). This suggests a simple counterexample.

Example 23 Consider the following system and state-wise attacker models:

- $\mathcal{SM}_{\mathsf{van}}$ is a two-state transition system, having an initial state $\hat{s}$ and another state $\hat{s}'$, and a single transition $\hat{s} \Rightarrow \hat{s}'$ (thus $\hat{s}'$ is final). $\mathcal{AM}_{\mathsf{van}}$ has no secrets or interaction (i.e., $\mathsf{isSec}_{\mathsf{van}}$ and $\mathsf{isInt}_{\mathsf{van}}$ are vacuously false).
- $\mathcal{SM}_{\mathsf{opt}}$ is the same as $\mathcal{SM}_{\mathsf{van}}$; however, to keep with the notations we used for unwinding, when referring to transitions in $\mathcal{SM}_{\mathsf{opt}}$ we will write s and s' instead of $\hat{s}$ and $\hat{s}'$. $\mathcal{AM}_{\mathsf{opt}}$ has $\mathsf{isSec}_{\mathsf{opt}}(s)$ true, and apart from this everything false, namely $\mathsf{isSec}_{\mathsf{opt}}(s')$, $\mathsf{isInt}_{\mathsf{opt}}(s)$ and $\mathsf{isInt}_{\mathsf{opt}}(s')$ false.

Clearly both relative security and finitary relative security hold for these systems (which are in fact each individually secure, since they have no interaction hence no leaks). However, there exists no unwinding (or finitary unwinding) relation Δ satisfying the hypotheses of Thm. 13, essentially because the unwinding conditions would force the secrets produced in the optimization-enhanced system to be matched by secrets produced in the vanilla system, which is impossible because the latter system produces no secrets. In detail, the "initial state coverage" hypothesis of Thm. 13 here means that $\Delta \infty (\infty, \infty)\ (s, s, \mathsf{eq})\ (\hat{s}, \hat{s}, \mathsf{eq})$ (or $\Delta \infty (s, s, \mathsf{eq})\ (\hat{s}, \hat{s}, \mathsf{eq})$ for finitary unwinding) holds. Moreover, from the definition of unwinding, we must have that:

- either **(i)** $\mathsf{react}(\Delta)\ (\infty, \infty)\ (s, s, \mathsf{eq})\ (\hat{s}, \hat{s}, \mathsf{eq})$ holds,
- or there exists $w_0 < \infty$ (i.e., $w_0 \in \mathbb{N}$) such that $\mathsf{proact}(\Delta)\ w_0\ (\infty, \infty)\ (s, s, \mathsf{eq})\ (\hat{s}, \hat{s}, \mathsf{eq})$ holds.

In the second case, by the definition of proact, we obtain $\Delta\ w_0\ (\infty, \infty)\ (s, s, \mathsf{eq})\ (\hat{s}_1, \hat{s}_2, \hat{st})$, for some $\hat{s}_1, \hat{s}_2$ and $\hat{st}$ such that $\hat{s}_1 = \hat{s}'$ and/or $\hat{s}_2 = \hat{s}'$ (i.e., at least one of $\hat{s}_1$ and $\hat{s}_2$ is $\hat{s}'$, since at least one vanilla-system transition must be taken in a proactive move and the only vanilla-system transition is from $\hat{s}$ to $\hat{s}'$), which again by the definition of unwinding, implies that:

- either **(ii)** $\mathsf{react}(\Delta)\ (\infty, \infty)\ (s, s, \mathsf{eq})\ (\hat{s}_1, \hat{s}_2, \hat{st})$ holds,
- or there exists $w_1 < w_0$ such that $\mathsf{proact}(\Delta)\ w_1\ (\infty, \infty)\ (s, s, \mathsf{eq})\ (\hat{s}_1, \hat{s}_2, \hat{st})$ holds.

Again in the second case, by the definition of proact, we obtain $\Delta\ w_1\ (\infty, \infty)\ (s, s, \mathsf{eq})\ (\hat{s}', \hat{s}', \hat{st}')$, for some $\hat{st}'$. Indeed, because at most one of $\hat{s}_1$ and $\hat{s}_2$ is different from $\hat{s}'$, and since again at least one vanilla-system transition must be taken in a proactive move, it must be the case that we reach $\hat{s}'$ on both vstate components. So by the definition of unwinding, this time only the reactive move is available, therefore obtaining **(iii)** $\mathsf{react}(\Delta)\ (\infty, \infty)\ (s, s, \mathsf{eq})\ (\hat{s}', \hat{s}', \hat{st}')$.

In any of these three cases, (i)–(iii), we obtain $\mathsf{react}(\Delta)\ (\infty, \infty)\ (s, s, \mathsf{eq})\ (\hat{s}_3, \hat{s}_4, \hat{st}'')$ for some $\hat{s}_3, \hat{s}_4$ and $\hat{st}''$. (The above reasoning, which we performed for unwinding, also works for finitary unwinding, removing the (∞, ∞) arguments.) But then, by the definitions of react and match^1, since $\neg\ \mathsf{isInt}_{\mathsf{opt}}(s)$, $s \Rightarrow s'$ and $\mathsf{isSec}_{\mathsf{opt}}(s)$ hold, it must be the case that the second or third disjunct in the definition of match^1 holds, in particular $\mathsf{eqSec}(s, \hat{s}_3)$ and therefore $\mathsf{isSec}_{\mathsf{van}}(\hat{s}_3)$ holds—which is impossible since $\mathsf{isSec}_{\mathsf{van}}$ is assumed vacuously false. So we have reached a contradiction, meaning that an unwinding relation (or a finitary unwinding relation) satisfying the hypotheses of Thm. 13 cannot exist. □

Other reasons why completeness of the unwinding proof method fails are that our notions of relative security and unwinding are extensions of the "absolute" (two-trace rather than four-trace) notions, which are known to be incomplete unless certain harsh assumptions are

made about the system and security models [25, 26]—which in turn is a reflection of the fact that the unwindings for the "absolute" notions can be encoded as automata simulations [26, §6] and therefore inherit the (in)completeness results from there [27].

As for our unwinding disproof method, the main reason why it is incomplete is also a form of requirement eagerness: For given sequences of secrets σ_1 and σ_2 (stemming from two chosen otraces π_1 and π_2), intuitively an SD unwinding ensures that any two vtraces $\hat{\pi}_1$ and $\hat{\pi}_2$ that produce the secrets σ_1 and σ_2 (i.e., $S_{\mathsf{van}}(\hat{\pi}_1) = \sigma_1$ and $S_{\mathsf{van}}(\hat{\pi}_2) = \sigma_2$) can *at no point* diverge on observations without diverging (yet) on actions—whereas relative security would allow them to diverge on observations earlier, provided they eventually also diverge on actions. This suggests the following counterexample to the completeness of the SD unwinding disproof method.

Example 24 Consider the following system and state-wise attacker models:

- $\mathcal{SM}_{\mathsf{van}}$ is a five-state transition system, having two initial states $\hat{s}_0$ and $\hat{s}'_0$ and another three states $\hat{s}_1, \hat{s}'_1, \hat{s}_2$, and has transitions $\hat{s}_0 \Rightarrow \hat{s}_1 \Rightarrow \hat{s}_2$ and $\hat{s}'_0 \Rightarrow \hat{s}'_1 \Rightarrow \hat{s}_2$ (thus $\hat{s}_2$ is final). $\mathcal{AM}_{\mathsf{van}}$ has:
  - $\mathsf{isSec}_{\mathsf{van}}(\hat{s}_0)$ and $\mathsf{isSec}_{\mathsf{van}}(\hat{s}'_0)$ true, and $\mathsf{isSec}_{\mathsf{van}}(\hat{s}_1)$, $\mathsf{isSec}_{\mathsf{van}}(\hat{s}'_1)$ and $\mathsf{isSec}_{\mathsf{van}}(\hat{s}_2)$ false; $\mathsf{getSec}_{\mathsf{van}}(\hat{s}_0) \neq \mathsf{getSec}_{\mathsf{van}}(\hat{s}'_0)$;
  - $\mathsf{isInt}_{\mathsf{van}}(\hat{s}_0)$, $\mathsf{isInt}_{\mathsf{van}}(\hat{s}'_0)$, $\mathsf{isInt}_{\mathsf{van}}(\hat{s}'_1)$ and $\mathsf{isInt}_{\mathsf{van}}(\hat{s}_1)$ true, and $\mathsf{isInt}_{\mathsf{van}}(\hat{s}_2)$ false; $\mathsf{getAct}_{\mathsf{van}}(\hat{s}_0) = \mathsf{getAct}_{\mathsf{van}}(\hat{s}'_0)$ and $\mathsf{getAct}_{\mathsf{van}}(\hat{s}_1) \neq \mathsf{getAct}_{\mathsf{van}}(\hat{s}'_1)$; $\mathsf{getObs}_{\mathsf{van}}(\hat{s}_0) \neq \mathsf{getObs}_{\mathsf{van}}(\hat{s}'_0)$.

  We let $\mathrm{sec}_0 = \mathsf{getSec}_{\mathsf{van}}(\hat{s}_0)$ and $\mathrm{sec}'_0 = \mathsf{getSec}_{\mathsf{van}}(\hat{s}'_0)$.
- $\mathcal{SM}_{\mathsf{opt}}$ is the same as $\mathcal{SM}_{\mathsf{van}}$; and again in the context of $\mathcal{SM}_{\mathsf{opt}}$ we will use "hat-free" notations, writing $s_0, s'_0, s_1, s'_1, s_2$ instead of $\hat{s}_0, \hat{s}'_0, \hat{s}_1, \hat{s}'_1, \hat{s}_2$. The state-wise attacker model $\mathcal{AM}_{\mathsf{opt}}$ has:
  - $\mathsf{isSec}_{\mathsf{opt}}(s_0)$ and $\mathsf{isSec}_{\mathsf{opt}}(s'_0)$ true, and $\mathsf{isSec}_{\mathsf{opt}}(s_1)$, $\mathsf{isSec}_{\mathsf{opt}}(s'_1)$ and $\mathsf{isSec}_{\mathsf{opt}}(s_2)$ false; $\mathsf{getSec}_{\mathsf{opt}}(s_0) = \mathrm{sec}_0$ and $\mathsf{getSec}_{\mathsf{opt}}(s'_0) = \mathrm{sec}'_0$;
  - $\mathsf{isInt}(s_0)$ and $\mathsf{isInt}(s'_0)$ true, and $\mathsf{isInt}_{\mathsf{opt}}(s_1)$, $\mathsf{isInt}_{\mathsf{opt}}(s'_1)$ and $\mathsf{isSec}_{\mathsf{opt}}(s_2)$ false; $\mathsf{getAct}_{\mathsf{opt}}(s_0) = \mathsf{getAct}_{\mathsf{opt}}(s'_0)$ and $\mathsf{getObs}_{\mathsf{opt}}(s_0) \neq \mathsf{getObs}_{\mathsf{opt}}(s'_0)$.

Both relative security and finitary relative security fail for these systems. This is because the otraces $\pi_1 = s_0\, s_1\, s_2$ and $\pi_2 = s'_0\, s'_1\, s_2$ are such that $\mathsf{A}_{\mathsf{opt}}(\pi_1) = \mathsf{A}_{\mathsf{opt}}(\pi_2)$, $\mathsf{O}_{\mathsf{opt}}(\pi_1) \neq \mathsf{O}_{\mathsf{opt}}(\pi_2)$, i.e, exhibit a leak of the secrets $\mathsf{S}_{\mathsf{opt}}(\pi_1) = \mathrm{sec}_0$ and $\mathsf{S}_{\mathsf{opt}}(\pi_2) = \mathrm{sec}'_0$. However, this leak cannot be reproduced in the vanilla system, where the only two vtraces, $\hat{\pi}_1 = \hat{s}_0\hat{s}_1\hat{s}_2$ and $\hat{\pi}_2 = \hat{s}'_0\hat{s}'_1\hat{s}_2$, have the property that $\mathsf{A}_{\mathsf{van}}(\hat{\pi}_1) \neq \mathsf{A}_{\mathsf{van}}(\hat{\pi}_2)$.

On the other hand, there is no SD unwinding satisfying the hypotheses of Thm. 18. Indeed, any such SD unwinding Γ would have to be based on the only two otraces, π_1 and π_2—more precisely on their produced singleton secret sequences sec_0 and sec'_0; so we would need to have $\Gamma\ (\hat{s}_0, \mathrm{sec}_0)\ (\hat{s}'_0, \mathrm{sec}'_0)$. Therefore, since $\mathsf{isInt}_{\mathsf{van}}(\hat{s}_0)$, $\mathsf{isInt}_{\mathsf{van}}(\hat{s}'_0)$ and $\mathsf{getAct}_{\mathsf{van}}(\hat{s}_0) = \mathsf{getAct}_{\mathsf{van}}(\hat{s}'_0)$ hold, by the definition of SD unwinding we would also have $\mathsf{getObs}_{\mathsf{van}}(\hat{s}_0) = \mathsf{getObs}_{\mathsf{van}}(\hat{s}'_0)$, which is false. □

Extensions of our notions of unwinding and SD unwinding, as well as the identification of suitable conditions on the system and attacker models, could address the problems with completeness that are either relative-security specific (such as the above pointed secrecy discrepancy problem for unwinding) or more fundamental, going back to automata simulations—the latter having a spectrum of limitations and partial solutions described in the literature [27]. We leave these as future work.

6 Language-Based Instantiation

We now move to demonstrating the applicability of our abstract definitions and methodologies for relative security in a concrete setting. We describe a concrete instance of relative security (§6.3), via a programming language with speculative execution (§6.1) that can express our running examples (§6.2).

6.1 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. [11], maintaining runtime configurations for nested speculative executions—and only for those that result from *misprediction*, since they are the only security-relevant ones. In particular, it will be able to express our running examples to a degree that is sufficiently faithful for capturing their relative (in)security.

Syntax. The set Lit of *literals*, ranged over by i, j etc., is taken to be $\mathbb{Z}$. 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:

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

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

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* $P = c_0; c_1; \ldots; c_n$ is a non-empty 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 $\mathbb{Z}$. Loc, the set of *locations*, ranged over by l, is also $\mathbb{Z}$.

Basic semantics. Fig. 10 shows the *basic (small-step) semantics*, denoted $\Rightarrow_B$, parameterized by a fixed program $P = c_0; \ldots; c_n$. 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) where:

- $pc \in \mathbb{N}$ is the *program counter (PC)*, i.e. c_{pc} is the pc'th statement in the program.
- 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;

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

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

$$\textsc{AVarAssign}\quad \frac{c_{pc} = (a[e] = e') \qquad \mu = (vs, avs, hp) \qquad avs(a) = (l, n) \qquad 0 \leq [\![e]\!](\mu) < n}{((pc, \mu), inp) \Rightarrow_B ((pc+1, \mu[(l + [\![e]\!](\mu)) \leftarrow [\![e']\!](\mu)]), inp)}$$

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

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

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

Fig. 10 Basic semantics for program $P = c_0; \ldots; c_n$. We implicitly assume $pc \leq n$ as a condition in each rule.

Fig. 11 Normal semantics for program $P = c_0; \ldots; c_n$. We implicitly assume $pc \leq n$.

$$\textsc{Standard}\quad \frac{(cfg, inp) \Rightarrow_B (cfg', inp')}{(cfg, inp, L) \Rightarrow_N (cfg', inp', L \cup \mathsf{readLocs}(cfg))}$$

- $avs : \mathsf{AVar} \to \mathsf{Loc} \times \mathbb{N}$ is an *array-variable store* assigning a heap starting location and size to each array-variable; and
- $hp : \mathsf{Loc} \to \mathsf{Val}$ is a *heap* assigning values to locations.

We let Mem denote the set of memories; thus, $\mathsf{Mem} = (\mathsf{Var} \to \mathsf{Val}) \times (\mathsf{AVar} \to \mathsf{Loc} \times \mathbb{N}) \times (\mathsf{Loc} \to \mathsf{Val})$ 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.g. $[\![e]\!](\mu)$. The *output of a configuration*, $\mathsf{outOf}(pc, \mu) \in (\mathsf{OCh} \times \mathsf{Val})_\perp$, is $(och, [\![e]\!](\mu))$ if c_{pc} has the form $\mathsf{Output}_{och}\ e$, and $\perp$ otherwise.

The read locations. To model Spectre vulnerabilities, we will record memory reads (as in [11]). We let $\mathsf{readLocs}(pc, \mu)$ be the (possibly empty) set of locations that are read (accessed) 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 $c[2] = a[b[3]]$, then $\mathsf{readLocs}$ returns two locations: counting from 0, the 2nd location of c, the 3rd location of b and the $b[3]$'th location of a.

Normal semantics. Fig. 11 shows the "normal" semantics $\Rightarrow_N$, which is the basic semantics extended to accumulate the read locations, hence account for cache side-channels.

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

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

$$\textsc{Mispred}\quad \frac{\begin{array}{c}\mathsf{isCond}(cfg_k) \qquad \mathsf{mispred}(ps, pcs) \\ (cfg_k, inp) \Rightarrow_B (cfg', inp') \qquad (cfg_k, inp) \Rightarrow_M (cfg'', inp'') \\ C' = cfg_0 \ldots cfg_{k-1} cfg' cfg'' \qquad L' = L \cup \mathsf{readLocs}(cfg_k)\end{array}}{(ps, cfg_0 \ldots cfg_k, inp, L) \Rightarrow_S (\mathsf{update}(ps, pcs), C', inp', L')}$$

$$\textsc{Resolve}\quad \frac{k > 0 \qquad \mathsf{resolve}(ps, pcs)\ \vee\ \mathsf{isIO}(cfg_k) \qquad C' = cfg_0 \ldots cfg_{k-1}}{(ps, cfg_0 \ldots cfg_k, inp, L) \Rightarrow_S (\mathsf{update}(ps, pcs), C', inp, L)}$$

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

Fig. 12 Speculative semantics for program $P = c_0; \ldots; c_n$ under misprediction oracle (PState, mispred, resolve, update). Each time, we implicitly assume $pc \leq n$ and $pcs = pc_0 \ldots pc_k$ where $pc_i = \mathsf{pcOf}(cfg_i)$ for $i \in \{0, \ldots, k\}$.

Speculative semantics. Finally, Fig. 12 shows the *speculative semantics* $\Rightarrow_S$, which augments the vanilla 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*, (PState, mispred, resolve, update) where:

- PState is a set of *predictor states* ranged over by *ps*;
- mispred : PState $\times \{0, \ldots, n\}^* \to$ Bool is the *misprediction* predicate;
- resolve : PState $\times \{0, \ldots, n\}^* \to$ Bool is the *resolution* predicate; and
- update : PState $\times \{0, \ldots, n\}^* \to$ PState is the *predictor-state update* function.

The oracle decides when misprediction occurs thus triggering a new speculation (the mispred predicate), and 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. 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 of speculation, i.e., the runtime environment determining that the prediction was wrong and reverting the corresponding speculative execution, can occur at any time after misprediction.

The speculative semantics $\Rightarrow_S$ operates on *(multi-speculative) states*, i.e., tuples $s = (ps, cfgs, inp, L)$ where $ps \in$ 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 transition that cfg_k can take is standard transitions, shown in the rule STANDARD. These occur only when **1)** the misprediction option is not available (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, such as the command being Fence, or an IO (input or output) command. Indeed, IO commands are not allowed to be executed speculatively, due to not being reversible.

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. 12, as the rule IFJUMPMISPRED. 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 or execution got stuck to an IO command, 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.

6.2 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 unspecified function in $\mathsf{Val} \times \mathsf{Val} \to \mathsf{Val}$.

Our motivating examples from §2 are expressed in IMP as depicted in Fig. 13. As expected, 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'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.

6.3 Instantiating Relative Security to IMP

We fix an IMP program $P = c_0; \ldots; c_n$, and predicates $\mathsf{imem} : \mathsf{Mem} \to \mathsf{Bool}$ and $\mathsf{ipstate} : \mathsf{Mem} \to \mathsf{Bool}$ identifying possible initial memories and predictor states. We first instantiate the system models $\mathcal{SM}_{\mathsf{van}}$ and $\mathcal{SM}_{\mathsf{opt}}$:

- $\mathsf{State}_{\mathsf{van}}$ consists of non-speculative states, (cfg, inp, L); $\Rightarrow_{\mathsf{van}}$ is the vanilla semantics, $\Rightarrow_N$; and $\mathsf{istate}_{\mathsf{van}}(cfg, inp, L)$ says that $cfg = (0, \mu)$ for some $\mu \in \mathsf{Mem}$ such that $\mathsf{imem}\ \mu$, and $L = \emptyset$.

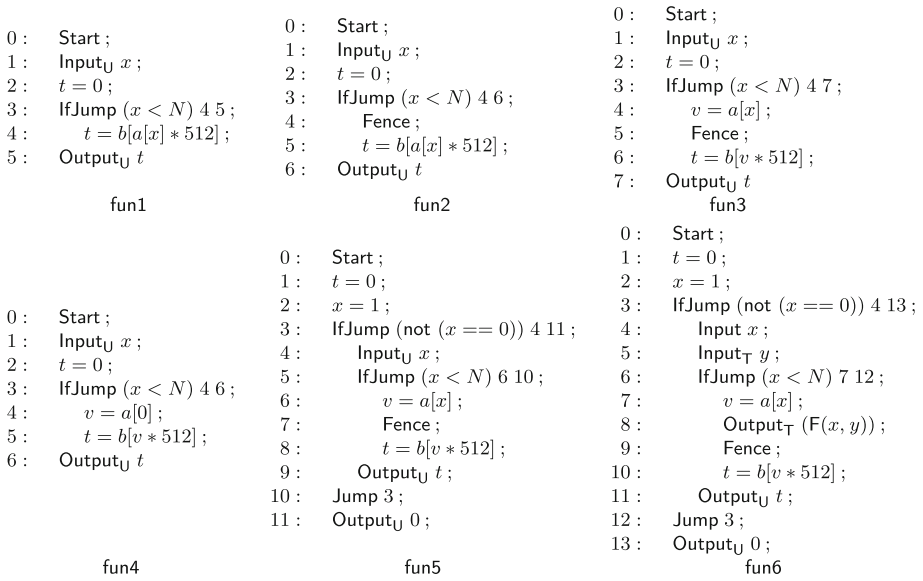


Fig. 13 Our six motivating examples from §2 written in IMP.

- $\mathsf{State}_{\mathsf{opt}}$ 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 $\mathsf{ipstate}$ ps, and $cfgs$ is a list consisting of a single configuration $(0, \mu)$ for some $\mu \in \mathsf{Mem}$ such that imem μ, and $L = \emptyset$.

Thus, both systems start with program counter 0, an initial memory and an input stream, and no read locations or speculation.

To instantiate 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, by taking the operators for $\mathcal{AM}_{\mathsf{van}}$ as the restrictions of those for $\mathcal{AM}_{\mathsf{opt}}$, we focus on describing the latter, while also omitting the subscript opt.

To determine isInt and getInt, we ask what the attacker's capabilities are. In addition to controlling/observing untrusted inputs/outputs, we assume a strong attacker observing execution time and control flow, following the leakage model of *constant-time security* [3]. 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 $s = (ps, cfg_0 \ldots cfg_k, inp, L)$, we have:

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

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

As shown by the definition of isInt, interaction occurs 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.

To determine isSec and getSec, we next ask 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. 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 pairs (A, B)

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.

In short, the secrets are now two-fold: memory if in the initial state, and trusted input (if any). With this instantiation, one can check that the intuitive (in)security of these examples discussed in §2 corresponds to if they (don't) satisfy relative security. As our informal analysis previously concluded, §8 demonstrates that why fun1 is not relatively secure, fun2–fun6 are.

7 Formally Instantiating Relative Security

In order to complete the proofs of relative security for our case studies, we create another Isabelle library [8] which extends on our earlier work (§4) to formally model our concrete setting (§6).

Fig. 14 presents the full Isabelle session graph for the concrete instantiation of relative security [8]. This builds on our `[Relative_Security]` session (which includes the unwinding), as well as the `[Secret_Directed_Unwinding]` session.

The formalization has four distinct parts, of which the first three are explored in this section: **1)** the IMP formalization (§7.1), including theories such as `Language_Syntax`, and `Step_Normal`, **2)** the relative security instantiation, i.e. any theory with the prefix `Instance` (§7.2), and then **3)** the formalization of each specific case study, in the `fun⟨i⟩` theories (§7.3). The fourth, which addresses the (dis)proofs of relative security for our examples, will be discussed in §8.

7.1 Formalizing the IMP Language

The theories `Language_Prelims`, `Language_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 language syntax definitions proceed similarly to previous formalizations of simple languages in Isabelle, such as in [28]. In addition to typical inclusions like arithmetic expressions (`aexp`) and command (`com`) datatypes, we additionally define other simple datatypes such as (`trustStat`)—which enables reasoning on subtle Spectre-like examples that require the existence of trusted and untrusted I/O channels. Other datatypes further model states, stores and configurations. There are also several supporting functions modeling useful syntax-based predicates and expression valuations.

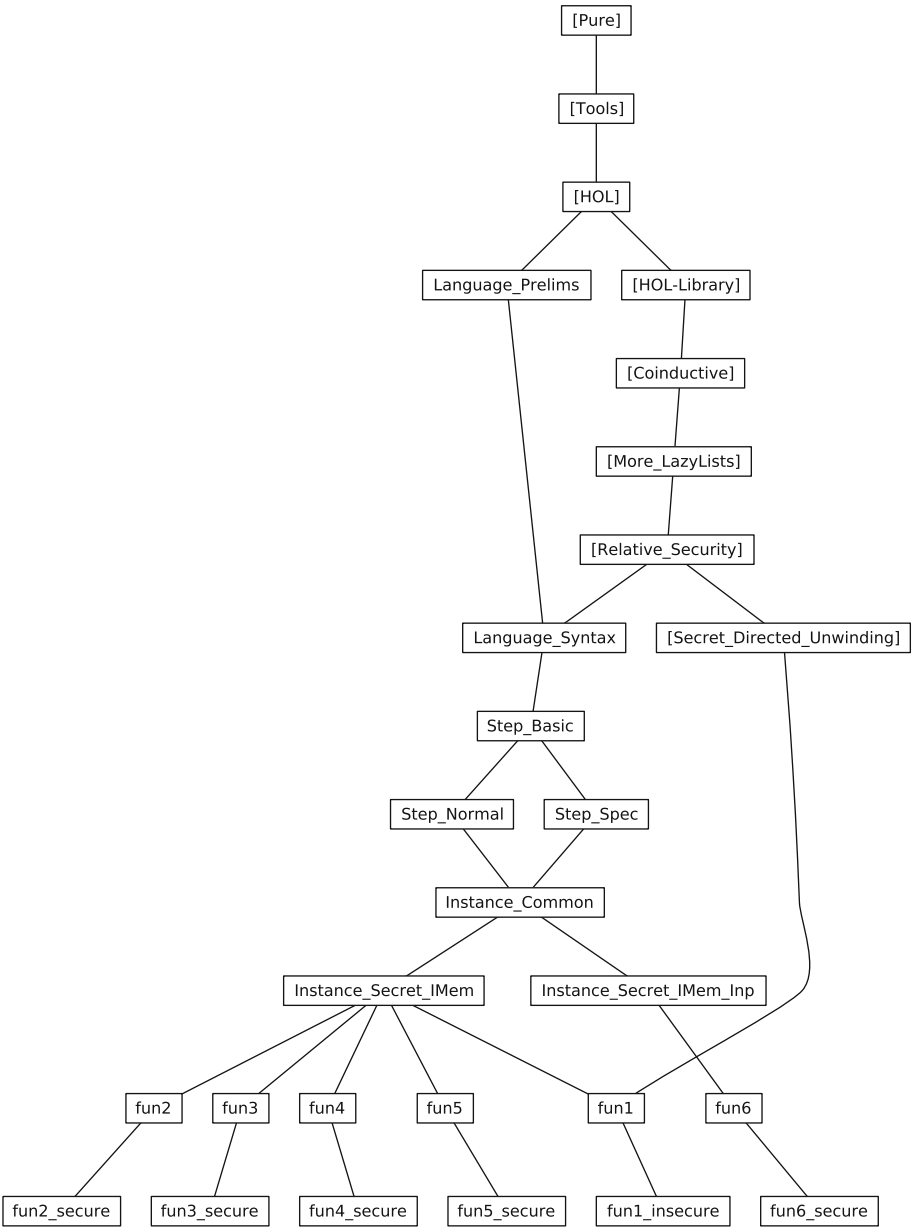


Fig. 14 Theory structure of our Isabelle mechanization.

We formalize the basic semantics of the language in the *Step_Basic* theory. Demonstrating another use of locales, this theory first sets up a locale *Prog*, to define a context for reasoning about well-formed programs.

```
locale Prog =
  fixes prog :: prog
  assumes wf_prog: "prog ≠ [] ∧ hd prog = Start"
```

Note that *prog* is used as both the name of the locale parameter, and the name of a type synonym for a list of commands. A program is defined to be well-formed if that list is non-empty and begins with *Start*.

We use an inductive definition, *stepB* ($\rightarrow_B$) to formalize the basic semantics within this locale context. Hence, the semantics are *parameterized* by a fixed program. The cases for the inductive definition follow those presented in (§6.1). This is supported by other simple local definitions for transition reasoning such as *finalB* and *nextB*.

Additionally, the locale context contains many basic lemmas on valid transitions of the semantics. These lemmas can be split into three classes.

- General lemmas on basic semantic properties.
- Simplification rules, which are sufficient conditions for a command to execute a transition to the next state.
- Elimination rules, which are useful for simplifying reasoning on preserving and updating invariants on the current state.

For example, the general lemma below demonstrates how we prove a basic deterministic property on our semantics using Isabelle.

```
lemma stepB_determ:
  "cfg_ib →B cfg_ib' ⟹ cfg_ib →B cfg_ib'' ⟹ cfg_ib'' = cfg_ib'"
  apply(induction arbitrary: cfg_ib'' rule: stepB.induct)
  by (auto elim: stepB.cases)
```

This proof applies induction using the generated induct rule from the *stepB* inductive definition. The resulting inductive cases can be simply discharged by applying *auto*.

The readLocs definition from (§6.1) is formalized using three functions that are independent of a locale context: *readLocsA* and *readLocsB* which compute the locations read by an arithmetic and boolean expression respectively, and *readLocsC* which uses these to compute the locations accessed by all commands. The resultant *readLocs* definition *within* the *Prog* context calls these functions on the fixed program *prog*.

The *Normal_Semantics* theory has a similar structure, but is much shorter. The *stepN* function (not shown) in Isabelle extends the existing *stepB* definition to include the read locations defined above. Additionally, we formalize similar definitions such as *finalN* and *nextN* to include this extension.

The speculative semantics outlined in (§6.1) are parameterized by both a program and a misprediction oracle. This is modeled by the *Prog_Mispred* locale, which directly inherits from our earlier *Prog* locale.

```
locale Prog_Mispred =  Prog prog
  for prog :: "com list"
  +
  fixes mispred :: "predState ⇒ pcounter list ⇒ bool"
  and resolve :: "predState ⇒ pcounter list ⇒ bool"
  and update :: "predState ⇒ pcounter list ⇒ predState"
```

Note that in the formalization, the set of predictor states, PState, from the misprediction oracle definition is modeled by an unspecified type *predState* (introduced earlier via **typedecl**), and as such there are only three new locale parameters. Within this locale, we formalize the *StepS* definition (not shown), as well as similar general, simp, and elimination lemmas for reasoning on transitions as before.

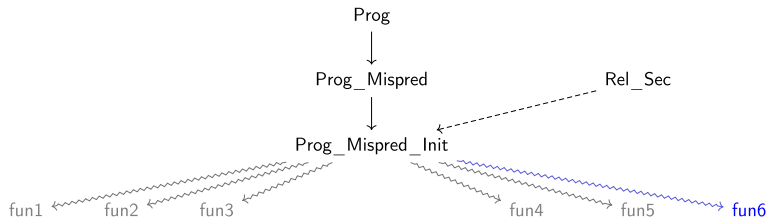


Fig. 15 Locale hierarchy for concrete programs.

7.2 Instantiating Relative Security

The instantiation of relative security to IMP, taking 0.6K LOC, is formalized in theories *Instance_Common*, *Instance_Secret_IMem*, and *Instance_Secret_IMem_Inp*.

The common theory covers the interaction infrastructure that is shared by all our motivating examples, such as the system models outlined in (§6.3). The formalization mirrors these definitions in two phases. Firstly, in the *Prog_Mispred* locale we use type synonyms and functions to set up our system model notation and concrete functions. For example, *getIntV* is an Isabelle function representing getInt(s) which takes a *stateV* (type synonym representing the state tuple) and returns a tuple $actV \times obsV$ representing the attacker's action (type synonym for a value) and its observation (type synonym for a tuple containing a value and read locations). Similar functions and type synonyms are set up for the optimized system with the *O* postfix.

The only common aspect left of the system model definition is the initialization. This uses another locale, *Prog_Mispred_Init*, which extends *Prog_Mispred* by introducing parameters for an initial prediction state, and initial state. The concrete definitions for the istate definitions are then defined within the locale as a function *istateV* and *istateO*.

The theories *Instance_Secret_IMem* and *Instance_Secret_IMem_Inp*, cover 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). The structure of both these theories is similar and proceeds as follows.

- First, we extend the *Prog_Mispred* locale by adding local definitions for *isSecV* and *isSecO* and functions for *getSecV* and *getSecO*.
- Second, we prove that the resultant *Prog_Mispred_Init* (which inherits from *Prog_Mispred*, including the new extensions) is a sublocale of our *Rel_Sec* locale from (§4) by instantiating all the parameters of the *Rel_Sec* locale with our concrete local definitions and functions.
- Finally, we set up a number of basic lemmas for reasoning about all the getter functions which are specific to the concrete instantiation.

The resulting locale structure is shown by the top half of Fig. 15, with the dotted line indicating the sublocale link to the *Rel_Sec* locale.

Importantly note that the *Instance_Secret_IMem* and *Instance_Secret_IMem_Inp* theories *do not* import each other, as indicated by Fig. 14. This is important as while both result in the same locale hierarchy structure as shown in Fig. 15, the *Prog_Mispred* locale in each file is extended with definitions that have similar names but different meanings depending on the secrecy infrastructure.

7.3 Formally Representing our Case Studies

Representing our case studies in the IMP language, taking about 0.7K LOC, mostly comprises "boilerplate" simplification rules (theories named *fun⟨i⟩* in Fig. 14).

The structure of each of these theories, and even some lemmas and proofs within them are, in general, similar. Hence, in this section we aim to present the common themes across each theory, with some examples used from the formalization of fun4.

A *fun⟨i⟩* theory begins by setting up some constants and definitions to represent basics such as array sizes and variable names. It then defines the following.

- Functions such as *initAvstore* and *istate*, where the initial store is defined to contain two arrays located one after the other.
- A definition *prog* which represents our program, matching the IMP representation in Fig. 13 from (§6.2).
- Functions *resolve*, *mispred*, *update*, and *initPstate*—in all proofs of relative security, just declared (unspecified) constants.
- Simplification lemmas on case splits (based on program counters).

With this setup in place, it is now possible to interpret the *Prog_Mispred_Init* locale—instantiating the locale parameters with these concrete definitions specific to a certain function.

```
interpretation Prog_Mispred_Init where
    prog = prog and initPstate = initPstate and mispred = mispred and
    resolve = resolve and update = update and istate = istate
  by (standard, simp add: prog_def)
```

This enables us to use all our earlier more abstract definitions in the concrete context of a particular program, noting that if a parameter was declared as a constant, it effectively still represents an arbitrary value in the interpretation. The interpretations are shown at the bottom of the locale hierarchy shown in Fig. 15. The different coloring is used to note that fun1-fun5 all use the version of the locale from *Instance_Secret_IMem*, whereas fun6 imports *Instance_Secret_IMem_Inp*.

With the definitions now in place, the theories proceed to setting up all the formal boilerplate for reasoning on basic properties of the program (not specific to relative security, although essential to those later proofs) as follows.

- Abbreviations to re-establish notation in the wider theory context (as an interpretation does not currently lift all locale notation with it).
- Simplification lemmas on input, output, fences, secrets, actions, and observations.
- Simplification lemmas for the *next*, *readLocs*, and *final* functions/predicates specific to the program and each stage of the program counter.

It is worth noting that the duplication between some of this "proof boilerplate" suggests two possible aspects of future work: **1)** automation of some boilerplate proofs, and **2)** development of a further intermediate abstraction level (using locales) which could enable some of these proofs and lemmas to be shared.

8 Unwinding (Dis)Proofs for our Examples

This section sketches the (dis)proofs of relative security for our six motivating examples as expressed in IMP (§6.2) using the unwinding technique developed in (§5).

The use of Isabelle was once again integral to these proofs, which are formalized in the *fun(i)*(in)secure theories involving approximately 1.3 LOC. In this section, we present the formal intuition behind the structure of each unwinding proof. We begin with the single insecure example and its formalization (§8.1), then focus on the secure examples (§8.2) and the structure of their formal proofs (§8.3).

8.1 Disproof for fun1

Intuitively fun1 is insecure because, for $x \geq N$ (which an attacker could input via the $\mathsf{Input}_{\mathsf{U}}\ x$ command), speculation can access the $(a[x]*512)$'th location of b whereas normal execution cannot. Formally, this is verified through the direct application or our disproof method (§5.7). To establish the traces, we choose a value $i \geq N$ as input for x (*vs1* in the formalization) and two initial arrays a that differ on $a[i]$ but nowhere else (defined using the *aa* variables in the formalization). These give two (singleton) sequences of secrets (the $\mathsf{S}(\pi_1)$ and $\mathsf{S}(\pi_2)$ in Thm. 18), and two otraces π_1 and π_2 that produce these secrets (i.e., start in memories with the respective arrays). The formalization defines these otraces as *s3_trans* and *s4_trans*, which, along with their respective starting states, are used to apply the formal disproof method *unwindSD_rsecure*.

theorem *"¬rsecure"*
  apply *(rule unwindSD_rsecure[of s3₀ s3_trans s4₀ s4_trans Γ_inv])*
    by *simp_all*

This proof is almost fully automatic, making use of several key lemmas that have been added to the *simp* set. Formally we're required to show Γ*_inv* is indeed an SD unwinding, and that it satisfies the initial states with respect to the secret sequences. The latter is straightforward, formalized by Γ*_init* with a mostly automated proof thanks to the basic helper lemmas on the traces. The main part of the formal proof comes with establishing the SD unwinding (*unwindSD*). While the formal proof appears relatively long, most proof steps could be easily discharged through the application of Isabelle's Sledgehammer thanks to the boilerplate put in place across earlier theories.

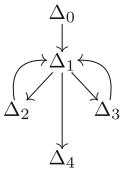
The structure of the proof is based on the following intuition. We must show that any two vtraces that produce these secrets (i.e., start in the two given memories) are observationally equal (i.e., output the same value) provided they are action-equal (i.e., input the same value x). Indeed, here a trivial secret-directed unwinding (defined formally by the Γ*_inv* function) 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.

8.2 Proofs for fun2-fun6

These proofs involve turning the informal intuition as to why these programs are (relatively) secure (see §2), into an unwinding argument over 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 $\hat{\pi}_2$).

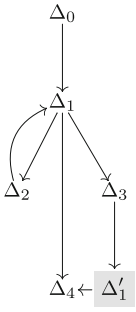
Crucially, all proofs refer to unwinding networks rather than single unwindings, and hence require Thm. 16. We have an unwinding component for each phase in the program execution (e.g., "before entering the if branch"), and we transit between components upon relevant events (e.g., "the if branch is taken"). Establishing how each component unwinds into the next (as required by the unwinding network definition, Def. 14) forms the majority of the

For fun2



	$(st, \hat{st})$	Spec.	$\hat{pc}_1 = \hat{pc}_2 = pc_1^0 = pc_2^0$	$pc_1^1 = pc_2^1$	Memory invariants	Read locs. invariants
Δ_0	(eq, eq)	No	0, 1	–	$\hat{\mu}_1 = \mu_1^0 \wedge \hat{\mu}_2 = \mu_2^0$	$\hat{L}_1 = L_1 \wedge \hat{L}_2 = L_2$
Δ_1	(eq, eq)	No	2–6	–	$\ldots \wedge \mu_1^0 =_x \mu_2^0$	…
Δ_2	(eq, eq)	Yes	4	6	…	…
Δ_3	(eq, eq)	Yes	6	4	…	…
Δ_4	(eq, eq) or (diff, diff)	No	7	–	…	…

For fun3



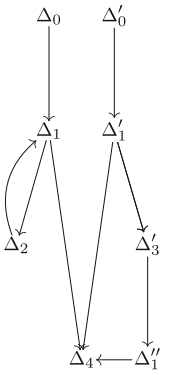
	$(st, \hat{st})$	Spec.	$\hat{pc}_1 = \hat{pc}_2 = pc_1^0 = pc_2^0$	$pc_1^1 = pc_2^1$	Memory invariants	Read locs. invariants
Δ_0	(eq, eq)	No	0, 1	–	$\hat{\mu}_1 = \mu_1^0 \wedge \hat{\mu}_2 = \mu_2^0$	$\hat{L}_1 = L_1 \wedge \hat{L}_2 = L_2$
Δ_1	(eq, eq)	No	2–7	–	$\ldots \wedge \mu_1^0 =_x \mu_2^0$	…
Δ_2	(eq, eq)	Yes	4	7	…	…
Δ_3	(eq, eq)	Yes	7	4, 5	$\ldots \wedge \mu_1^1 =_x \mu_2^1$	$(pc_1^1 = 4 \longrightarrow \ldots) \wedge d(L_1, L_2) \subseteq d(\hat{L}_1, \hat{L}_2)$
Δ_1'	(eq, eq)	No	7	–	same as for Δ_1	…
Δ_4	(eq, eq) or (diff, diff)	No	8	–	…	…

Fig. 16 Summary of the unwinding proofs for the relative security of fun2, fun3, and fun4 when $N > 0$.

proof, and is in turn crucial to structuring the formalization. As such, this section focuses on these unwinding aspects, with the main discussion of the formalization left for (§8.3).

Examples of the unwinding networks used by these arguments for fun2-fun4 are given in Figs. 16 and 17. On the left, the figure shows the network as a graph which represents how the unwinding may transition to various states (omitting loops). On the right, the figures show a table summary of what the relations Δ_i say about the statuses $st, \hat{st}$ and the states $s_1, s_2, \hat{s}_1, \hat{s}_2$. Recall each state has a configuration for each speculation level (a PC and a memory), and the set of locations set so far, which the table addresses separately. The columns represent: **1)** the status, **2)** if the program is in a speculative state, **3)** PC values (and input properties), **4)** memory invariants, and **5)** read location invariants. We use the following notation.

- pc_i^j is the PC at speculation level j from the configuration that is part of the ostate, s_i, where $i \in \{1, 2\}$.
- For the memory μ_i^j, we use superscript to indicate the speculation level.
- For the vstates, we do not have anything beyond level 0 (as no speculation can occur), so we do not use any superscripts in $\hat{pc}_i$ or $\hat{\mu}_i$ (for $i \in \{1, 2\}$).
- The sets of read locations for the otraces and vtraces are denoted L_i and $\hat{L}_i$, respectively.
- $\mu_1^i =_x \mu_2^i$ means that the memories μ_1^i and μ_2^i have the same value for the variable x; by contrast, $\mu_1^i =_x^{\neg} \mu_2^i$ means that the two memories have the same value for everything except possibly for x.
- "–" means "not applicable" or "vacuously true", and "…" refers to the invariant from the previous line.



	$(st, \hat{st})$	Spec.	$(\hat{pc}_1 = \hat{pc}_2, pc_1^0 = pc_2^0, pc_1^1 = pc_2^1)$	Input properties	Memory invariants	Read locs. invariants
Δ_0	(eq, eq)	No	$(0,0,-)$ or $(1,1,-)$	$i_1 = i_2 = \hat{i}_1 = \hat{i}_2$	$\hat{\mu}_1 = \mu_1^0 \wedge \hat{\mu}_2 = \mu_2^0$	$\hat{L}_1 = L_1 \wedge \hat{L}_2 = L_2$
Δ_1	(eq, eq)	No	$(2,2,-)$ or $\vdots$ or $(6,6,-)$	–	$\ldots \wedge \mu_1^0 =_x \mu_2^0 \wedge \mu_1^0(x) < N$	$\ldots$
Δ_2	(eq, eq)	Yes	$(4,4,6)$	–	$\ldots$	$\ldots$
Δ'_0	(eq, eq)	No	$(0,0,-)$ or $(1,1,-)$	$i_1 = i_2 \wedge \hat{i}_1 = \hat{i}_2 = 0$	$\hat{\mu}_1 = \mu_1^0 \wedge \hat{\mu}_2 = \mu_2^0$	$\hat{L}_1 = L_1 \wedge \hat{L}_2 = L_2$
Δ'_1	(eq, eq)	No	$(2,2,-)$ or $\vdots$ or $(6,6,-)$	–	$\ldots \wedge \mu_1^0 =_x \mu_2^0 \wedge \mu_1^0(x) \geq N \wedge \hat{\mu}_1(x) = \hat{\mu}_2(x) = 0$	$\ldots$
Δ'_3	(eq, eq)	Yes	$(4,6,4)$ or $(4,6,5)$	–	$\ldots \wedge \mu_1^1 =_x \mu_2^1$	$(pc_1^1 = 4 \longrightarrow \ldots) \wedge d(L_1, L_2) \subseteq d(\hat{L}_1, \hat{L}_2)$
Δ''_1	(eq, eq)	No	$(4,6,-)$ $(5,6,-)$ $(6,6,-)$	–	$\ldots$	$\ldots$
Δ_4	(eq, eq) or (diff, diff)	No	$(7,7,-)$	–	$\ldots$	$\ldots$

Fig. 17 Summary of the unwinding proofs for the relative security of fun2, fun3, and fun4 when $N > 0$.

We will refer to Figs. 16 and 17 throughout the remainder of this section.

Common aspects of the proofs. 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 any speculation level j (i.e., $pc_1^j = pc_2^j$). The same is true for the two vstates $\hat{s}_1$ and $\hat{s}_2$ (i.e., $\hat{pc}_1 = \hat{pc}_2$). Moreover, often (but not always) the vstates have the same PC with the level-0 PC of the ostates, such as in fun2 and fun3: $\hat{pc}_1 = \hat{pc}_2 = pc_1^0 = pc_2^0$ 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 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.

The initial secrets of s_1 and s_2 are allowed to differ, i.e., in the values stored in the arrays a and b. According to the secrecy contract, $\hat{s}_1$ should always 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) must ensure this, whenever an output statement gets executed, i.e. if the output 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 the key goal of each proof, forming the observation part of the interaction contract. In our examples,

all programs have only a single Output command (occurring at the end), so this is the only possible time observations may differ.

Lastly, 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.

From a formal perspective, some of these commonalities are built into the setup outlined in (§7.2), as well as the similarities between the function formalizations outlined in (§7.3). Additionally, the `fun⟨i⟩`(in)secure theories often share elements of their boilerplate definitions (such as program counter setup).

The proof for fun2. This 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.

Formally, as outlined by the first table in Fig. 16, we have a single initial relation, Δ_0, which covers PCs 0 and 1 (namely those before the input is taken). The first row of the table shows as expected that the PCs are synchronized, the corresponding state memories are equivalent ($\hat{\mu}_i = \mu_i^0$), and the read locations are equivalent ($\hat{L}_i = L_i$). The equalities are maintained in all rows of the table, and ensure that the otrace observations differ if and only if the vtrace observations do—as an observation consists of a value in memory and the set of read locations.

The relations Δ_1–Δ_4 account for all possible situations in fun2 after the input has been taken.

- Δ_1 for when no speculation occurs.
- Δ_2 for when speculation occurs and mispredicts the "else" branch—i.e. normal execution (at level 0) takes the "then" branch (PC 4) and speculative execution (at level 1) takes the "else" branch and gets stuck at the Output command (PC 6).
- Δ_3 for when speculation occurs and mispredicts the "then" branch—i.e. normal execution takes the "else" branch (PC 6) and speculative execution takes the "then" branch and gets stuck at the Fence (PC 4).
- Δ_4 for when the program is finished (hence the PC is at 7, past the last command).

Transitions are possible from Δ_1 to Δ_2 or Δ_3 when at PC 3, the only place where speculation can occur. Conversely, Δ_2 can transit back to Δ_1 when speculation gets resolved, and Δ_3 can transit back to Δ_1 when either speculation gets resolved or the Fence command is executed. Finally, only Δ_1 can transit to the exit relation Δ_4, since the Output command is the only exit point of the program, and cannot be executed speculatively.

The proof for fun3. This proof is slightly more subtle than fun2. In the only interesting scenario, speculation incorrectly takes the "then" branch. As the Fence is placed after a location read, the otraces can access locations that the vtraces cannot access, namely the x'th location of a. The goal of our unwinding proof is to show this is harmless—more specifically that the otraces are not observationally "more different" than the vtraces are.

Unlike fun2, the invariant $\hat{L}_i = L_i$ can no longer be maintained for all Δ_i. However, it suffices to make sure that $\hat{L}_1$ and $\hat{L}_2$ stay "at least as different as" L_1 and L_2—which we express by the invariant $d(L_1, L_2) \subseteq d(\hat{L}_1, \hat{L}_2)$, where $d(A, B)$ is the distance (also known as the symmetric difference) between two sets A and B, defined as $(A \setminus B) \cup (B \setminus A)$.

In Fig. 16, we highlight the differences between the proof of fun3 and that of fun2 occurring in the Δ_3 row representing the "then" branch misprediction. This shows that speculation can be at PCs 4 or 5. Speculatively executing the assignment at PC 4 increases L_i while the vanilla $\hat{L}_i$ does not change. This would break the original invariant $\hat{L}_i = L_i$, however still satisfies our distance invariant. To support this invariant's preservation within Δ_3, *before* speculatively executing the assignment the following conditions had to hold: **i)** the original invariant, $\hat{L}_i = L_i$, and **ii)** the same location reads by two otrace transitions, $\mu_1^1 =_x \mu_2^1$.

The proof for fun4. This function is relatively secure when assuming $N > 0$. The proof requires use of two unwinding strategies depending on whether the value of the input is $< N$ or not. While this distinction was made in the proofs for fun2 and fun3 as well (distinguishing between Δ_2 for "else" branch misprediction when $x < N$ and Δ_3 for "then" branch misprediction), here we must act on it earlier when the Input command is being executed. This can clearly be seen in Fig. 17, where two initial relations are used for fun4 in the unwinding network, and we additionally consider input properties in the table. In fact, comparing the unwinding networks for fun3 and fun4, we see that the latter has been obtained by splitting the former along Δ_0 and Δ_1 to create variants of these relations based on the initial split.

The first case is not problematic, because its corresponding otraces can only diverge from vtraces in an immediately harmless way, i.e., by (mis)speculating on the "else" branch. This requires a similar proof strategy to our earlier proofs, as can be seen in the first three rows of the table.

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 array b depending on the value of $a[0]$ (on which they may differ in the speculative memories μ_1^1 and μ_2^1). As such the sets of read locations L_1 and L_2 may differ. To counter this, we choose a different input for the vtraces, namely 0 (as indicated by the memory invariants column in the table for Δ_1'). This means that the vtraces also take the "then" branch, keeping their PCs synchronized with the level-1 PCs of the otraces.

We now have two subcases.

1. The otraces engage in the speculation, resulting in a transition to Δ_3'. In this case we react to the synchronous steps taken by the speculation, and when a resolution step occurs in the otraces, transition to Δ_1''. Here, the vtraces continue to take one or two proactive execution steps on the "then" branch, thus ensuring: **(i)** that the program counters synchronize on 6, ready to transit to the exit relation, **(ii)** that the outputs of the otraces are both equal to 0, and **(iii)** that our difference invariant stating the sets of read locations of the vtraces are at least as different as those of the otraces, $d(L_1, L_2) \subseteq d(\hat{L}_1, \hat{L}_2)$ given the vtraces have certainly executed the entire "then" branch.
2. The otraces do not engage in speculation. Here, we still proactively execute the commands on the "then" branch so the vtraces are once again ready to execute the output command in sync with the otraces while maintaining the difference invariant.

The proof of fun5. For the case of fun5, a slight modification of the strategy for fun3 is used, catering for extra speculative cases introduced by the while loop. As before, we have a single initial relation of Δ_0, which covers PCs 0, 1 and 2 and unwinds into Δ_1.

From Δ_1, there are two points where speculation can occur. Firstly, at the "while" condition (PC 3), we unwind into Δ_1' which covers both mispredicting cases:

- normal execution takes the "then" branch (PC 4) and speculation (at level 1) takes the "else" branch (PC 11)

- normal execution takes the "else" branch (PC 11) and speculation (at level 1) takes the "then" branch (PC 4)

In both situations the otraces get stuck at the output (PC 11) and input (PC 4) respectively, since these cannot be executed speculatively, both transit back into Δ_1 upon resolution.

Secondly, at the "if" condition (PC 5), relations Δ_2 and Δ_3 from the proof of fun3 maintaining almost the same transitions as before. The key difference is that Δ_2 can reach second level of speculation at the "while" condition (PC 3), but this once again is harmless, as the same mispredicting cases used for Δ_1' can be discharged. Additionally, Δ_3 also transits to Δ_1 (instead of some variant) when speculation is resolved.

This same strategy is maintained indefinitely until $x = 0$. If $x = 0$, then after one final execution of the loop PC $4 - 10$ (using the same fun3 strategy) and PC = 11 is reached Δ_1 transits to the exit relation Δ_4.

The proof of fun6. 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.

8.3 Structure of the Formal Proofs

Unsurprisingly, the structure of the formal proofs is very similar across case studies. Thus, the theories provide a blueprint for how to apply this approach to potential future examples. Each theory begins by setting up a *common* definition, which encapsulates properties common to all unwinding relations in the proof. Elements of this definition are the same across all case studies.

The formalization then defines each unwinding in the network, such as *Δ0*, which each extend the common relation with the required additional properties, e.g. as represented in the tables in Figs. 16 and 17.

For each unwinding, a lemma such as *Δ0_implies* is shown via a straightforward proof to establish facts on elements of the program execution such as the program counter and store which that unwinding relation implies.

This setup leads to the main part of the proof. Firstly, the *init* lemma shows the initial states satisfy the initial condition, as required by the definition of the unwinding network. Then, a step lemma for each unwinding, such as *step0*, establishes the *UnwindIntoCond* required to prove they form an unwinding network. Below are examples from the *Fun3_secure* theory.

```
lemma init: "initCond Δ0"
lemma step0: "unwindIntoCond Δ0 (oor Δ0 Δ1)"
```

Each proof is guided by the formal intuition outlined in (8.2). For example, the *step0* lemma above states that *Δ0* can unwind into either *Δ0* or Δ1. While these proofs are relatively long, many of the proof goals could be automatically proven via application of sledgehammer, and were easily structured to begin with by applying rules such as case splits on the speculative semantics. As such, the real challenge in these proofs comes with establishing what the unwinding network looks like and the intuition behind the transitions. After establishing this, the formal proof mostly follows (and likely could even be further automated in future work).

The final step in the formalization is to establish relative security, the last proposition in each Isabelle theory. This proof locally defines *m*, *nxt* and Δs (mirroring the n, next and $\overline{\Delta}$ elements of the unwinding tuple in Def. 14. Application of the *distrib_unwind_rsecure* proof method (i.e. Thm. 16), results in four subgoals which are discharged automatically using the above outlined lemmas. We give the example from *Fun3_secure* below.

```
proposition rsecure
  proof –
    define m where m: "m ≡ (6::nat)"
    define Δs where Δs: "Δs ≡ λi::nat.
    if i = 0 then Δ0
    else if i = 1 then Δ1
    else if i = 2 then Δ2
    else if i = 3 then Δ3
    else if i = 4 then Δ4
    else Δ1'"
    define nxt where nxt: "nxt ≡ λi::nat.
    if i = 0 then {0,1::nat}
    else if i = 1 then {1,2,3,4}
    else if i = 2 then {1}
    else if i = 3 then {3,5}
    else {4}"
    show ?thesis apply(rule distrib_unwind_rsecure[of m nxt Δs])
      subgoal unfolding m by auto
      subgoal unfolding nxt m by auto
      subgoal using init unfolding Δs by auto
      subgoal
        unfolding m nxt Δs apply (simp split: if_splits)
        using theConds
        unfolding oor_def oor4_def by auto .
  qed
```

Note that only fun5 and fun6 require application of the *distrib_unwind_lrsecure* lemma (for possibly infinitary unwinding), due to the while loops. However, this has practically no impact on any other part of the case study proofs, as all the hard work was done in the abstract soundness proof in (§5).

9 Related Work

Of the vast literature on Spectre and Meltdown [29], we will only discuss the formal modeling and verification aspects. In particular, this section focusses on the contract/policy pattern (§9.1), a more detailed comparison to TPOD [11] (§9.2) and a view of these comparisons in the light of controlled declassification (§9.3), as well as discussing some related work in the formalization setting (§9.4) and on other proof approaches (§9.5).

9.1 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 A). Notably, this is the case of Guanciale et al.'s *conditional non-interference* [30], an extension of noninterference [16, 31] 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*

noninterference [14] (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. [3], which, borrowing concepts and terminology from Guarnieri et al. [32], 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 $[\![\cdot]\!]^\alpha_l : \mathsf{State} \to \mathsf{Seq}(\mathsf{Obs})$. 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 $\forall s_1, s_2 \in \mathsf{State}.\ s_1 \simeq_p s_2 \longrightarrow [\![s_1]\!]^\alpha_l = [\![s_2]\!]^\alpha_l$. 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 $\forall s_1, s_2 \in \mathsf{State}.\ s_1 \simeq_p s_2 \wedge [\![s_1]\!]^{\alpha'}_{l'} = [\![s_2]\!]^{\alpha'}_{l'} \longrightarrow [\![s_1]\!]^\alpha_l = [\![s_2]\!]^\alpha_l$.

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 $[\![\cdot]\!]^\alpha_l$, 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.

9.2 TPOD

Cheang et al.'s *TPOD (trace-property dependent observational nondeterminism)* [11] is an exception to the above rule, in that it explicitly captures both active attackers *and* interactive uploading of the secrets. TPOD extends observational determinism [17] from a two-trace to a four-trace property.

It is parameterized by a notion of low-equivalence $\equiv_{\mathsf{low}}$ on states, describing 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. Low equivalence and the high/low operations are extended component-wise from states to traces. Finally, rather than considering two 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):

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

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. By contrast, our design of relative security, in the more concrete attacker-model setting which we also deployed for the concrete examples, requires that the similarities between the vtraces and the otraces should refer to the underlying secrets, not to the attacker-taken actions. This is consistent with the standard assumption that the attacker is free to take *any* actions to exhibit a leak. On the other hand, depending on the context, there can be value in requiring the same attacker actions, as we illustrate in §9.3.

Interestingly, this over-constraining aspect of TPOD is illustrated by an example in the TPOD paper itself [11], namely the conditionally vulnerable program we showed in Fig. 4 (Fig. 3(c) in [11]): 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 [11]), TPOD requires that the vtraces reproduce the leak with the same input—which is impossible because, as we explained in §2 (following a discussion from [11]), 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, Fig. 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, the strength of TPOD is that it enables successful automatic verification which deems secure several interesting programs (as elaborated in [11]).

9.3 Controlled Declassification

As already pointed out, in its attacker-model-based instance (§3.3), our notion of relative security allows the reproducibility of a leak in the optimization-enhanced system to happen via different actions than the ones from the original leak in the vanilla system— which seems like a natural choice in light of examples such as Fig. 4 (the same as Fig. 3(c) in [11]).

```
1 void fun7(unsigned p, unsigned x)
      {
2   unsigned t = 0;
3   if (p == getSecretPassword()) {
4     t = b[a[x] * 512];
5   }
6 }
```

Listing 7 Relatively secure program

On the other hand, consider the program fun7 in Listing 7, where we assume the inputs p and x are controlled by the attacker, and the values of the array a are secret. Relative security, in its attacker-model instantiation,

would deem this program secure, because any leak of the value $a[x]$ through speculative execution can be replicated through normal execution by the attacker choosing p to be the correct password. By contrast, TPOD (as discussed in §9.2, seemingly against the intention of its authors) would deem it insecure because, for a combination of inputs involving the wrong password, a leak occurs speculatively which cannot be replicated non-speculatively using the same inputs. And for essentially the same reason, the notions we surveyed in §9.1, including conditional and speculative noninterference, would also deem this program insecure—assuming we encode the attacker-controlled inputs as a public part of the initial state in those models (which lack an explicit notion of action/input).

Since it deems the program (relatively) secure by counting on the attacker to guess the password in the non-speculative case, here our relative security instantiation is problematic. The crux of the problem is that our (attacker-model-based) notion, while allowing different actions in the two leaks that are being matched, does not also compare the *difficulty* or *(un)likelihood* of these leaks.

As discussed in Example 6, we can also instantiate our "very abstract" notion of relative security (§3.2, based on leakage models) to require coincidence on the actions as well. In future work, we will extend our (dis)proof techniques to leakage models that are more general than attacker models, which will allow us to flexibly factor in declassification [15, 33–36], as well as the gap between active and passive attackers, previously modeled in the literature as *robust declassification* [37, 38]. Indeed, the above difference that the attacker actions can make in establishing the (relative) security of systems stems precisely from their lack of robustness in the sense of [37]. Robustness is studied there at the same level of abstraction as our state-wise attacker model based relative security, which would facilitate a possible future combination of these two frameworks.

9.4 Formalization Related Work

Of the related work discussed in (§9.1) and (§9.2), none appear to have used a proof assistant to mechanize their results initially. However, a mechanization is now available in Isabelle of TPOD due to work by Griffin and Dongol [39]. Comparatively, in this work we used Isabelle to develop the proofs alongside the development of our theory.

In the broader field of information-flow security research, there are a number of notable examples of theories backed by substantial mechanizations across several proof assistants. Major verification projects specific to information-flow include a hardware architecture with information-flow primitives [40] and a separation kernel [41] in Coq, and noninterference for seL4 in Isabelle [42]. There have been a number of explorations of non-interference in a concurrent setting with formalizations in Isabelle [43–46].

Our formalization does not depend on any existing semantics or security formal developments in Isabelle. It is perhaps worth noting that our IMP language formalization is different from the existing IMP related libraries in the Isabelle distribution [28], as it additionally aims to capture speculation.

9.5 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 §9). Proof-theoretically, the framework generalizes the unwinding method [4], which was widely deployed in the security literature [25, 31, 47–49]. 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 [50–53]. Basing automatic analysis on unwinding has happened in the past with simpler notions of information-flow security, e.g., Volpano and Smith [50], Sabelfeld and Sands [51] and Boudol and Castellani [52] all base the soundness of their type systems for concurrent noninterference on a notion of bisimulation (from the unwinding family). Popescu et al. [54, 55] give a unified overview of this process. Developing such automated methods on top of our interactive proof-theoretic foundation could recover and perhaps generalize existing work based on model checking [11], symbolic execution [14] and type systems [53]—for example, Guarnieri et al. [14] list as future work the extension of their analysis to also cover non-terminating programs.

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 [56–58]. Relative security seems related to Morgan's refinement order for noninterference [59, 60]; working out the formal connection between the two notions could lead to more insights into compositional proofs for our notion.

10 Conclusion

This paper introduced relative security, a general model-theoretic and proof-theoretic foundation for expressing the information-flow security in the presence of semantics optimizations. Our main motivation came from Spectre-like vulnerabilities enabled by speculative and out-of-order execution, but our framework can in principle be applied more broadly, e.g., to compiler optimizations [61].

We presented the process of abstractly building our definition (§3), with the key novelty of capturing fully interactive attackers and dynamic secrets, along with the development of novel proof theoretic unwinding tactics to reason on key properties (§5). We then demonstrated how both the definition and proof methods can be applied on concrete examples (§6 and §8).

As highlighted throughout the paper (including in §4 and §7), the Isabelle proof assistant was a vital tool not only in proving the results, but also in developing the relevant concepts. Notably, obtaining a sound yet sufficiently general unwinding proof method required several iterations, with amending the unwinding conditions based on the feedback we received from Isabelle while attempting (and failing) the soundness proof, or when attempting (and failing) the proof of a relevant example. During these activities, we had the distinct feeling that

using a proof assistant such as Isabelle may be the *only* way to stay on top of the emerging complexity. The involved Isabelle libraries are all publicly available on the Archive of Formal Proofs [6–8].

Acknowledgements We thank the CSF and JAR 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)". Dongol is additionally supported by EPSRC grants EP/Y036425/1, EP/X037142/1, EP/V038915/1, EP/R025134/2 and VeTSS. Popescu is additionally supported by the EPSRC grant EP/V039156/1 ("Security of Digital Twins in Manufacturing").

Author Contributions AP and BD designed the abstract framework, based on previous discussions between these authors and JD, and with some input from JW and MG. AP and JW designed the unwindinding proof and disproof methods and proved their soundness. JW formalized the programming language semantics and proved the case studies of (in)secure programs. MG provided some insight into the concrete case studies based on his work on proving security for C programs. CE provided a restructuring of the formalization to make better use of Isabelle's locales. The text of the paper was mainly written by AP, BD, JW and CE. All authors participated in discussions about most of the aspects of this paper, and reviewed the manuscript. The authors are listed in alphabetical order, regardless of contributions or seniority.

Data Availability No datasets were generated or analysed during the current study.

Declarations

Competing interests The authors declare no competing interests.

References

1. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M.: Meltdown: Reading kernel memory from user space. In: USENIX Sec. (2018)
2. Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre attacks: Exploiting speculative execution. In: SP, pp. 1–19. IEEE, USA (2019). https://doi.org/10.1109/SP.2019.00002
3. Cauligi, S., Disselkoen, C., Moghimi, D., Barthe, G., Stefan, D.: Sok: Practical foundations for software spectre defenses. In: SP, pp. 666–680. IEEE, USA (2022). https://doi.org/10.1109/SP46214.2022.9833707
4. Goguen, J.A., Meseguer, J.: Unwinding and inference control. In: SP, pp. 75–87 (1984)
5. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Springer, Heidelberg (2002)
6. Popescu, A., Wright, J.: Relative security. Archive of Formal Proofs (2024). https://isa-afp.org/entries/Relative_Security.html, Formal proof development
7. Dongol, B., Griffin, M., Popescu, A., Wright, J.: Secret-directed unwinding. Archive of Formal Proofs (2024). https://isa-afp.org/entries/Secret_Directed_Unwinding.html, Formal proof development
8. Wright, J., Popescu, A.: A formalized programming language with speculative execution. Archive of Formal Proofs (2024). https://isa-afp.org/entries/IMP_With_Speculation.html, Formal proof development
9. Dongol, B., Griffin, M., Popescu, A., Wright, J.: Relative security: Formally modeling and (dis)proving resilience against semantic optimization vulnerabilities. In: 2024 IEEE 37th Computer Security Founda-

tions Symposium (CSF), pp. 403–418. IEEE Computer Society, Los Alamitos, CA, USA (2024). https://doi.org/10.1109/CSF61375.2024.00027
10. Kocher, P.: Spectre Mitigations in Microsoft's C/C++ Compiler. https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html (2018)
11. Cheang, K., Rasmussen, C., Seshia, S.A., Subramanyan, P.: A formal approach to secure speculation. In: CSF, pp. 288–303. IEEE, USA (2019)
12. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Com. Sec. **18**(6), 1157–1210 (2010)
13. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Sel. Areas in Comm. 21(1), 5–19 (2003)
14. Guarnieri, M., Köpf, B., Morales, J.F., Reineke, J., Sánchez, A.: Spectector: Principled detection of speculative information flows. In: SP, pp. 1–19. IEEE, USA (2020)
15. Popescu, A., Bauereiss, T., Lammich, P.: Bounded-deducibility security (invited paper). In: Cohen, L., Kaliszyk, C. (eds.) 12th International Conference on Interactive Theorem Proving (ITP 2021). LIPIcs, vol. 193, pp. 3–1320. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2021). https://doi.org/10.4230/LIPICS.ITP.2021.3
16. Goguen, J.A., Meseguer, J.: Security policies and security models. In: SP, pp. 11–20 (1982)
17. Zdancewic, S., Myers, A.C.: Observational determinism for concurrent program security. In: IEEE CSF Workshop, pp. 29–43 (2003)
18. The Isabelle theorem prover. http://isabelle.in.tum.de/ (2013)
19. Kammüller, F., Wenzel, M., Paulson, L.C.: Locales—a sectioning concept for Isabelle. In: TPHOLs, pp. 149–166 (1999)
20. Ballarin, C.: Tutorial to Locales and Locale Interpretation. In: Contribuciones Científicas en Honor de Mirian Andrés Gómez, pp. 123–140. University of Rioja, Spain (2010). Online at https://dialnet.unirioja.es/servlet/articulo?codigo=3216664
21. Ballarin, C.: Locales: A module system for mathematical theories. J. Autom. Reason. **52**(2), 123–153 (2014). https://doi.org/10.1007/S10817-013-9284-7
22. Lochbihler, A.: Coinductive. Archive of Formal Proofs (2010). http://afp.sourceforge.net/entries/Coinductive.shtml, Formal proof development
23. Popescu, A., Wright, J.: More operations on lazy lists. Archive of Formal Proofs (2024). https://isa-afp.org/entries/More_LazyLists.html, Formal proof development
24. Sangiorgi, D.: On the bisimulation proof method. Math. Struc. Com. Sci. **8**(5), 447–479 (1998)
25. Mantel, H.: A uniform framework for the formal specification and verification of information flow security. PhD thesis, University of Saarbrücken (2003)
26. Mantel, H.: Unwinding possibilistic security properties. In: Cuppens, F., Deswarte, Y., Gollmann, D., Waidner, M. (eds.) Computer Security - ESORICS 2000, 6th European Symposium on Research in Computer Security, Toulouse, France, October 4-6, 2000, Proceedings. Lecture Notes in Computer Science, vol. 1895, pp. 238–254. Springer, Berlin, Heidelberg (2000). https://doi.org/10.1007/10722599_15
27. Lynch, N.A., Vaandrager, F.W.: Forward and backward simulations: I. untimed systems. Inf. Comput. 121(2), 214–233 (1995) https://doi.org/10.1006/INCO.1995.1134
28. Nipkow, T., Klein, G.: Concrete Semantics: With Isabelle/HOL. Springer, Switzerland (2014)
29. Canella, C., Bulck, J.V., Schwarz, M., Lipp, M., Berg, B., Ortner, P., Piessens, F., Evtyushkin, D., Gruss, D.: A systematic evaluation of transient execution attacks and defenses. In: USENIX Sec., pp. 249–266 (2019)
30. Guanciale, R., Balliu, M., Dam, M.: Inspectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis. In: CCS, pp. 1853–1869 (2020)
31. Rushby, J.: Noninterference, transitivity, and channel-control security policies. Technical report, SRI (Dec 1992). http://www.csl.sri.com/papers/csl-92-2/
32. Guarnieri, M., Köpf, B., Reineke, J., Vila, P.: Hardware-software contracts for secure speculation. In: SP, pp. 1868–1883 (2021). https://doi.org/10.1109/SP40001.2021.00036
33. Banerjee, A., Naumann, D.A., Rosenberg, S.: Expressive declassification policies and modular static enforcement. In: 2008 IEEE Symposium on Security and Privacy (SP 2008), 18-21 May 2008, Oakland, California, USA, pp. 339–353. IEEE Computer Society, USA (2008). https://doi.org/10.1109/SP.2008.20
34. Sabelfeld, A., Sands, D.: Declassification: Dimensions and principles. J. Comput. Secur. **17**(5), 517–548 (2009)
35. Askarov, A., Myers, A.C.: A semantic framework for declassification and endorsement. In: Gordon, A.D. (ed.) Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6012, pp. 64–84. Springer, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11957-6_5

36. Mantel, H., Reinhard, A.: Controlling the what and where of declassification in language-based security. In: Programming Languages and Systems. LNCS, vol. 4421, pp. 141–156 (2007)
37. Zdancewic, S., Myers, A.C.: Robust declassification. In: 14th IEEE Computer Security Foundations Workshop (CSFW-14 2001), 11-13 June 2001, Cape Breton, Nova Scotia, Canada, pp. 15–23. IEEE Computer Society, USA (2001). https://doi.org/10.1109/CSFW.2001.930133
38. Myers, A.C., Sabelfeld, A., Zdancewic, S.: Enforcing robust declassification and qualified robustness. J. Comput. Secur. **14**(2), 157–196 (2006). https://doi.org/10.3233/JCS-2006-14203
39. Griffin, M., Dongol, B.: Isabelle files for "Verifying Secure Speculation in Isabelle/HOL". (2021). https://figshare.com/s/c185541c43a7cac258b6
40. Amorim, A.A., Collins, N., DeHon, A., Demange, D., Hriţcu, C., Pichardie, D., Pierce, B.C., Pollack, R., Tolmach, A.: A verified information-flow architecture. In: POPL, pp. 165–178 (2014)
41. Dam, M., Guanciale, R., Khakpour, N., Nemati, H., Schwarz, O.: Formal verification of information flow security for a simple ARM-based separation kernel. In: CCS, pp. 223–234 (2013)
42. Murray, T.C., Matichuk, D., Brassil, M., Gammie, P., Bourke, T., Seefried, S., Lewis, C., Gao, X., Klein, G.: seL4: From general purpose to a proof of information flow enforcement. In: IEEE Symposium on Security and Privacy, pp. 415–429 (2013)
43. Barthe, G., Nieto, L.P.: Formally verifying information flow type systems for concurrent and thread systems. In: FMSE, pp. 13–22 (2004)
44. Popescu, A., Hölzl, J., Nipkow, T.: Formal verification of language-based concurrent noninterference. J. Formaliz. Reason. **6**(1), 1–30 (2013). https://doi.org/10.6092/issn.1972-5787/3690
45. Murray, T., Sison, R., Pierzchalski, E., Rizkallah, C.: Compositional Verification and Refinement of Concurrent Value-Dependent Noninterference. In: 2016 IEEE 29th Computer Security Foundations Symposium (CSF), pp. 417–431. IEEE, Lisbon (2016). https://doi.org/10.1109/CSF.2016.36 . https://ieeexplore.ieee.org/document/7536391/ Accessed 2024-03-25
46. Winter, K., Coughlin, N., Smith, G.: Backwards-directed information flow analysis for concurrent programs. In: 2021 IEEE 34th Computer Security Foundations Symposium (CSF), pp. 1–16. IEEE, Dubrovnik, Croatia (2021). https://doi.org/10.1109/CSF51468.2021.00017 . https://ieeexplore.ieee.org/document/9505223/ Accessed 2024-03-25
47. McLean, J.: A general theory of composition for trace sets closed under selective interleaving functions. In: SP, pp. 79–93 (1994)
48. Murray, T.C., Matichuk, D., Brassil, M., Gammie, P., Klein, G.: Noninterference for operating system kernels. In: CPP, pp. 126–142 (2012)
49. Popescu, A., Lammich, P., Hou, P.: CoCon: A conference management system with formally verified document confidentiality. J. Autom. Reason. **65**(2), 321–356 (2021). https://doi.org/10.1007/S10817-020-09566-9
50. Volpano, D., Smith, G.: Probabilistic noninterference in a concurrent language. Journal of Computer Security 7(2,3), 231–253 (1999)
51. Sabelfeld, A., Sands, D.: Probabilistic noninterference for multi-threaded programs. In: IEEE Computer Security Foundations Workshop, pp. 200–214 (2000)
52. Boudol, G., Castellani, I.: Noninterference for concurrent programs and thread systems. Theor. Comp. Sci. **281**(1–2), 109–130 (2002)
53. Shivakumar, B.A., Barthe, G., Grégoire, B., Laporte, V., Oliveira, T., Priya, S., Schwabe, P., Tabary-Maujean, L.: Typing high-speed cryptography against spectre v1. In: SP, pp. 1094–1111. IEEE, USA (2023). https://doi.org/10.1109/SP46215.2023.10179418
54. Popescu, A., Hölzl, J., Nipkow, T.: Proving concurrent noninterference. In: Hawblitzel, C., Miller, D. (eds.) Certified Programs and Proofs - Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7679, pp. 109–125. Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-35308-6_11
55. Popescu, A., Hölzl, J., Nipkow, T.: Formalizing probabilistic noninterference. In: Gonthier, G., Norrish, M. (eds.) Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings. Lecture Notes in Computer Science, vol. 8307, pp. 259–275. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-03545-1_17
56. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: FM. LNCS, vol. 6664, pp. 200–214 (2011). https://doi.org/10.1007/978-3-642-21437-0_17
57. Godlin, B., Strichman, O.: Regression verification: proving the equivalence of similar programs. Softw. Test. Verif. Reliab. **23**(3), 241–258 (2013). https://doi.org/10.1002/STVR.1472
58. Hur, C., Neis, G., Dreyer, D., Vafeiadis, V.: The power of parameterization in coinductive proof. In: POPL, pp. 193–206. ACM, USA (2013). https://doi.org/10.1145/2429069.2429093
59. Morgan, C.: The shadow knows: Refinement and security in sequential programs. Sci. Comput. Program. **74**(8), 629–653 (2009)

60. Morgan, C.: Compositional noninterference from first principles. Form. Asp. Comput. **24**(1), 3–26 (2012). https://doi.org/10.1007/S00165-010-0167-Y
61. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1998)