Non-interference Preserving Optimising Compilation

JULIAN ROSEMANN, Saarland University, Saarland Informatics Campus, Germany
SEBASTIAN HACK, Saarland University, Saarland Informatics Campus, Germany
DEEPAK GARG, Max Planck Institute for Software Systems, Saarland Informatics Campus, Germany

To protect security-critical applications, secure compilers have to preserve security policies, such as non-interference, during compilation. The preservation of security policies goes beyond the classical notion of compiler correctness which only enforces the preservation of the semantics of the source program. Therefore, several standard compiler optimisations are prone to break standard security policies like non-interference. Existing approaches to secure compilation are very restrictive with respect to the compiler optimisations that they permit or to the security policies they support because of conceptual limitations in their formal setup.

In this paper, we present *hyperproperty simulations*, a novel framework to secure compilation that models the preservation of arbitrary k-hyperproperties during compilation and overcomes several limitations of existing approaches, in particular it is more expressive and more flexible. We demonstrate this by designing and proving a generic non-interference preserving code transformation that can be applied on different optimisations and leakage models. This approach reduces the proof burden per optimisation to a minimum. We instantiate this code transformation on different leakage models with various standard compiler optimisations that could be handled in a very limited and less modular way (if at all) by existing approaches. Our results are formally verified in the Rocq theorem prover.

CCS Concepts: • **Security and privacy** → **Formal methods and theory of security**; • **Software and its engineering** → **Compilers**; • **Theory of computation** → *Logic and verification.*

Additional Key Words and Phrases: non-interference, secure compilation, hyperproperty preservation

ACM Reference Format:
Julian Rosemann, Sebastian Hack, and Deepak Garg. 2025. Non-interference Preserving Optimising Compilation. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 323 (October 2025), 26 pages. https://doi.org/10.1145/3763101

1 Introduction

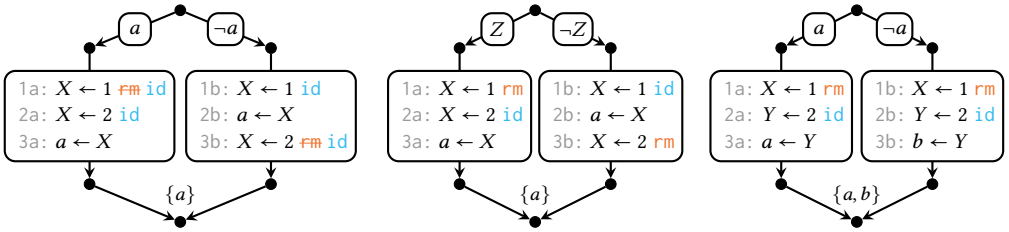
The preservation of security policies goes beyond the classical notion of compiler correctness which only enforces the preservation of the program's semantics. In contrast to the semantics, which is defined by individual traces, many security properties are characterized by two or more traces. For example, indistinguishability[1]-based properties like non-interference are characterized by two traces while robust declassification [Cecchetti et al. 2017] as well as absence of speculative side-channel leaks [Cheang et al. 2019] are characterized by four traces. Additionally, attacker models usually allow for more observations than what is considered observable behaviour by the semantics (e.g. cache accesses, memory addresses, timing information). Thus, if the source program

[1]As the term *observational equivalence* is used for both compiler correctness and specific security policies, we use the term *indistinguishability* for the latter to avoid ambiguity.

Authors' Contact Information: Julian Rosemann, Saarland University, Saarland Informatics Campus, Saarbrücken, Saarland, Germany, jtrosemann@cs.uni-saarland.de; Sebastian Hack, Saarland University, Saarland Informatics Campus, Saarbrücken, Saarland, Germany, hack@cs.uni-saarland.de; Deepak Garg, Max Planck Institute for Software Systems, Saarland Informatics Campus, Saarbrücken, Saarland, Germany, dg@mpi-sws.org.

ACM 2475-1421/2025/10-ART323
https://doi.org/10.1145/3763101

fulfils some desired security policy, classical compiler correctness is not enough to guarantee the preservation of this security policy, and indeed several standard compiler optimisations (e.g. dead code elimination) are prone to break standard security policies. Compilation with the aim to preserve security policies is called *secure compilation*.



(a) Dead-code elimination can destroy non-interference in secret branches.

(b) Dead-code elimination on a *public* branch preserves non-interference.

(c) Our approach is able to prove non-interference preservation for dead-code elimination on *secret* branches when possible.

Fig. 1. Dead-code elimination in three different example programs. The set at the join denotes the live-out set. Statements marked with rm are eliminated by dead code elimination, and so is their leakage. We use id to mark statements whose leakage is preserved through compilation. All other statements are not leaky.

Consider the example in Fig. 1a and assume that lower-case letters denote *secret* and upper-case letters denote *public* variables. As a simple example leakage model, we assume that the attacker can observe the values written to *public* variables. The program as-is exhibits non-interference which here means that the attacker cannot deduce the value of the secret input variable a from what they can observe on any two executions with equal *public* inputs: Regardless of whether a or $\neg a$ holds, the leakage trace is always $[X \leftarrow 1, X \leftarrow 2]$. A *correct* compiler might soundly eliminate the *dead* assignments at lines 1a and 3b. However, the resulting program is not non-interfering: For $a = \mathit{true}$ the leakage trace is $[X \leftarrow 2]$, while for $a = \mathit{false}$ the leakage trace is $[X \leftarrow 1]$. Thus, the attacker can deduce the value of a which means standard dead code elimination can break non-interference.

Now, assume a different leakage model where the attacker is only capable of observing *which* variable is written but not its value: Here, both the untransformed program (leakage traces are $[X, X]$) and the transformed program (leakage traces are $[X]$) exhibit non-interference. Consequently, the choice of the leakage model influences whether a transformation preserves non-interference.

Existing Approaches. One approach to secure compilation is to directly relate source and target leakage: The Jasmin compiler [Almeida et al. 2017] accompanies any compilation step with a global[2] *leakage transformer*: a *function* f that takes a source leakage event and returns a target leakage event. Assuming equal leakage in the source this implies equal leakage in the target, as any target leakage event l' equals $f\, l$ for a corresponding source leakage l.

Although sufficient for a non-optimising compiler, this approach is prohibitively strict: Fig. 1b shows that there is no global leakage transformer for dead code elimination. Dead code elimination removes $X \leftarrow 1$ on the left branch while keeping the assignment to X on the right branch. Thus, there is *no* target leakage on the left at that point while on the right the target leakage is still the *same* as in the source program. This discrepancy cannot be reflected by a global leakage transformer.

[2]As we will later introduce *local* leakage transformer we call the originally proposed leakage transformer *global*.

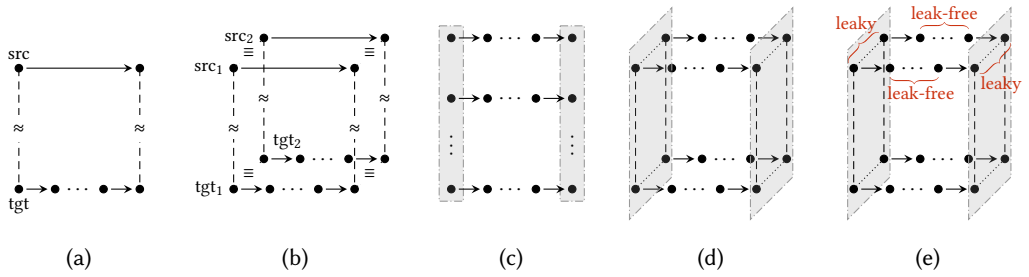


Fig. 2. (a) A simulated step between a source and a target trace: Any step in the source trace is simulated by multiple steps in the target trace, preserving some *simulation relation* as invariant. (b) A CT-simulation diagram where a single source step on two source traces is simulated by multiple steps in the two target traces. (c) A k-simulation, the dash-dotted boxes denote the sets of related states across k traces. (d) A hyperproperty simulation. (e) An indistinguishability simulation.

But in this example the transformations *are* preserving non-interference: Since the branch predicate is *public*, we know that the two traces considered for non-interference both take the same branch and thus both target traces exhibit the same leakage. Similar examples exist for all optimisations that change the leakage based on the location of the modified code.

To support more optimisations, the setup of classical simulation proofs (see Fig. 2a) has been extended to simulations of source and target trace *pairs:* Barthe et al. introduce CT-simulations, allowing for simulation proofs of the preservation of cryptographic constant-time[3] (in short: CT) [Barthe et al. 2018]. The intuition is visualised in Fig. 2b: Source and target traces are replaced by two pairs of traces, which correspond to the two traces with *public*-equal inputs. The CT-simulation now consists of three simulations: A (lockstep) simulation between the source traces, a (lockstep) simulation between the target traces, and a (less restricted) simulation between each source trace and its related target trace with equal inputs—on each computation step one has to ensure that all these relations are preserved. Several optimisations have been proven to preserve CT using this technique [Barthe et al. 2020, 2018; Shivakumar et al. 2022].

However, lockstep simulations forbid any branches on secrets, and thus CT-simulations cannot be used for preserving non-interference in the presence of secret branches (as in Fig. 1c). Although CT is predominant, recent work has shown that retaining secret branches and using a secure control flow balancing can be beneficial for performance [Winderix et al. 2024]. To the best of our knowledge there is no proof of non-interference for a flow-sensitive optimisation on a leakage model that allows (some) secret branches. Additionally, CT-simulations are complex to work with because one has to prove a simulation on *two* source and *two* target traces. For instance, in prior work [Barthe et al. 2020] that discusses the implementation of CT-simulations in CompCert, the authors mention that they use CT-simulations as sparsely as possible.

Our Contribution. In this paper, we present *hyperproperty simulations* that are more expressive and more modular than CT-simulations. First, our technique does not require that all branch predicates are *public* like CT-simulations. Second, our technique allows for modularising the proofs such that the proof burden per optimisation is just a specific simulation proof on *one* source and *one* target trace. To this end, we introduce several generalisations of simulations that build on each other (see Fig. 2). First, we generalise standard stuttering simulations (a)[4] (acting on *one* source and *one* target trace) to k-simulations acting on an arbitrary number (k) of traces (c). Here, steps are as

[3]Note that CT can be interpreted as a special case of non-interference, where in particular any branching decision is leaked.
[4]Sometimes also called *well-founded* or *"star"* simulations.

unrestricted as possible: For every step from a related set of related states to another, any trace may make an arbitrary number of steps, but there has to be a guarantee of *eventual progress on all traces*. As a generalisation to CT-simulations and as a special case of k-simulations, we introduce *hyperproperty simulations* (d): These consist of a (k-)source relation, a (k-)target relation and a relation between a source and a target trace. Together, these form a $2k$-simulation in the above sense. This combination of different simulations is crucial for the modularisation of the proofs. Finally, to model indistinguishability properties such as non-interference, we introduce *indistinguishability simulations* (e) as another special case of a hyperproperty simulations: We exploit the assumption of non-interference in the source traces to *jump* from a pair of related leakage events to the next, allowing for a concise proof argument. This approach even works in the presence of secret branches, making hyperproperty simulations more expressive than CT-simulations.

Consider Fig. 1c for an illustration of our non-interference preservation argument: We assume non-interference on the source traces, thus leakage events always occur in form of pairs. It is sufficient to show that on every such pair the source leakage equality implies target leakage equality. At the lines $(1a, 1b)$ the leakage is eliminated in the target traces at both program points, and at the lines $(2a, 2b)$ the target traces will have the *same* leakage as the source traces at that point—which is equal by assumption. Since these are the only occurrences of leakage, the transformation preserves non-interference. In our approach such an annotation of a *local leakage transformer* (`id` or `rm` in this case) is the only thing we need for proving non-interference preservation of an optimisation. Coming up with such local leakage transformers is typically straightforward.

Our approach is general enough to handle CT and simplifies CT proofs significantly: We show that in the absence of secret branches (which is a prerequisite for CT) any optimisation that adheres to a local leakage transformer annotation preserves non-interference. Adherence to a local leakage transformer annotation means that the leakage transformers exactly describe the leakage of the target trace. Showing the adherence to the local leakage transformers is a source-target simulation proof which typically is significantly simpler than CT-simulation proofs which involve four traces. Consider the example program in Fig. 1b. All traces with *public*-equal inputs will take the same branch direction. Let us assume they take the one on the right. Then, in the leakage model of CT which leaks all branch targets, the pairs of leakage events are $(0, 0)$ (0 indicates the branch predicate, leaking "right") and $(1b, 1b)$, and $(3b, 3b)$ (both leaking "write on X"). Being at the same program point implies having the same local leakage transformer which implies CT preservation.

Finally, we introduce NIFTY[5], a non-interference preserving code transformation framework. NIFTY provides a generic proof of indistinguishability that is parametrised by an optimisation, a leakage model, and a proof of consistency of the corresponding local leakage transformers. As mentioned above, this proof is a simple source-target simulation in contrast to a more complicated CT-simulation. NIFTY's core idea is to optimise code inside secret branches aggressively, but check—using the local leakage transformer annotations—at every control flow rejoin whether the optimisation violated indistinguishability. If so, NIFTY rewinds the optimisation for this code section and uses a less-aggressive fallback pass. As it was illustrated in Fig. 1a, it depends on the leakage model how many branches are backtracked.

We prove NIFTY to be non-interference preserving using an indistinguishability simulation. The proof holds for any leakage model. The optimisation has to provide data flow transformers generating a local leakage transformer annotation and a proof that the target leakage indeed adheres to this annotation. Additionally a fallback pass with a global leakage transformer is required—for many optimisations this is just the identity function.

[5]NIFTY is short for "**n**on-**i**nterference **f**acilitating **t**ransformations for **y**our optimisations".

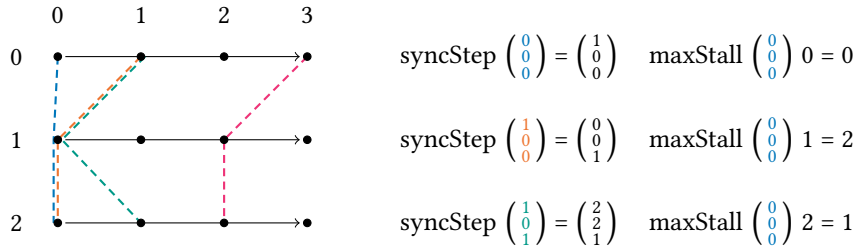


Fig. 3. Three traces with synchroniser functions acting on it. The states connected by dashed lines are related.

We apply NIFTY on dead code elimination, common branch factorisation, and a linearisation pass. The latter is similar to the only pass in [Barthe et al. 2020] using a CT-simulation and whose CT proof was described as a "not especially pleasant" by the authors. Due to the much more expressive language used there, a direct comparison is not meaningful. Our technique could be used to simplify the proof in their setup as well: Using local leakage transformers, the simulation on two source and two target traces could be reduced to just a simulation on a source and a target trace.

To summarise, our contributions are:

- We develop a novel framework for hyperproperty preservation proofs. We exploit that many relevant hyperproperties can be seen as simulations and introduce the notion of *k-simulations* to express simulations on an arbitrary number of traces. We show how to *compose* k-simulations to construct complex simulations from simpler ones. Based on this, we introduce *hyperproperty simulations* to compose hyperproperty preservation proofs from a source-target compilation simulation and a simulation on the source traces.
- We instantiate this general framework to prove the preservation of indistinguishability properties such as non-interference. We capture this in the notion of *indistinguishability simulation* which is a hyperproperty simulation that consists of a source-target leakage simulation and indistinguishability on the source traces. This way, our approach allows for separating compilation-specific arguments from security policy specific arguments which is not directly possible in related work. This setup simplifies cryptographic constant-time preservation proofs significantly and enables, for the first time, proofs of more general notions of non-interference.
- Many compiler optimisations are in general not non-interference preserving. To address this, we present the non-interference preserving code transformation framework *NIFTY*. NIFTY can be instantiated with an optimisation and a leakage model. NIFTY checks the results of the optimisation and undoes them in code regions where the optimisation broke non-interference. We prove NIFTY correct independent of the concrete optimisation and leakage model, assuming that a source-target simulation proof for the specific optimisation and leakage model is given.

Our contributions are formalised and proven correct in the Rocq theorem prover.

2 Key Ideas

This section gives an overview over the key ideas of our approach. First, we give an intuition for the different simulation notions that we introduce. Based on these notions, we sketch how proofs of CT-preservation can be reduced to a simple source-target simulation proof. Finally, we explain the core idea of our code transformation framework NIFTY, allowing for non-interference preservation even in the presence of *secret* branches.

k-simulations and synchronisers. A standard simulation proof (see Fig. 2a) essentially ensures an invariant between a source and a target trace with each step. If the simulation at some point kept *stalling* on the target trace, we could not prove anything for the remaining trace. Therefore, it is necessary to guarantee *progress* in the target. The standard approach is to use a ranking function that decreases whenever one trace *stalls* [Namjoshi 1997].

We introduce a new technique to apply this idea to an arbitrary number k of traces, where we have to guarantee *progress on all traces*. We use *lane indices* of type $\mathbb{N}_{<k}$ to identify individual traces and *positions* of type $\mathbb{N}^k$ to point to a specific point in each trace, i.e. the natural number represents how many steps have been processed since the initial state. We synchronise k traces using a pair of functions $(\text{syncStep}, \text{maxStall})$ acting on positions and lane indices: Consider Fig. 3. The function $\text{syncStep} : \mathbb{N}^k \to \mathbb{N}^k$ returns for every position the number of steps any individual lane does until the next synchronised position is reached. For example, at position $(1,0,0)$ it returns $(0,0,1)$, thus the next synchronised position is $(1,0,1)$. For any specific lane $\text{maxStall} : \mathbb{N}^k \to \mathbb{N}_{<k} \to \mathbb{N}$ denotes how many synchronised steps the lane is allowed to *stall* until there is progress. For example, at position $(0,0,0)$ lane 1 is allowed to stall twice, while lane 2 is only allowed to stall once and lane 0 is not allowed to stall at all. If syncStep observes the restrictions given by maxStall, *eventual progress on all lanes* is guaranteed, i.e. there is an infinite stream of synchronised positions. We then call $(\text{syncStep}, \text{maxStall})$ a (k)-*synchroniser*.

Assuming a k-tuple of traces and a relation $\approx$, we say $\approx$ is a k-*simulation* if there is a k-synchroniser, such that $\approx$ holds on the start states (at position 0^k) and is preserved over any synchronised step from a synchronised position to another (see Fig. 2c).

Combining k-simulations to hyperproperty simulations. This definition of a k-simulation allows us to *combine* several k-simulations, which is crucial for the modularisation of the non-interference preservation proof. Figures 4a and 4b give the basic intuition for this step (with $k = 2$): We assume a k-simulation on the source traces (depicted on top in (a), horizontally) and for any source trace a simulation to its target trace (depicted below in (a), vertically). The k-simulation on the source traces can now be *extended* to the target traces by matching equal positions on the source traces and fetching the associated target position from the source-target simulation on the specific trace (see Fig. 4b), resulting in a $2k$-simulation. Existing techniques for hyperproperty simulations need to consider all k traces at once which can complicate proofs significantly. Our approach allows us to prove the invariant between a source and a target trace and the invariant between two source traces separately, and then combine them to conclude the invariant on the two target traces, reducing the proof effort.

Indistinguishability simulations. We apply the idea from the previous paragraph to build *indistinguishability simulations* from two orthogonal components: The horizontal component is the indistinguishability of the source traces and the vertical component is the local leakage transformer simulation between a source and a target trace.

We say two traces t_0, t_1 are indistinguishable if for any finite prefix of t_0 there is a finite prefix of t_1 such that both emit the same sequences of leakages and vice versa. In other words, the leakage of one trace is *simulated* on the other[6]. We can model this as a 2-simulation: The simulation relation is *leakage equality* and we define the synchroniser using a case distinction:[7]

- either there is further leakage on both traces, then it *jumps* to that position,
- or there is no further leakage *ever*, in which case it is defined as $(1,1)$.

[6]This captures the notion of "coproductive security" as defined in [Bohannon et al. 2009].
[7]In general, this synchroniser is not computable. But this is not a problem because we only need it as a proof argument.

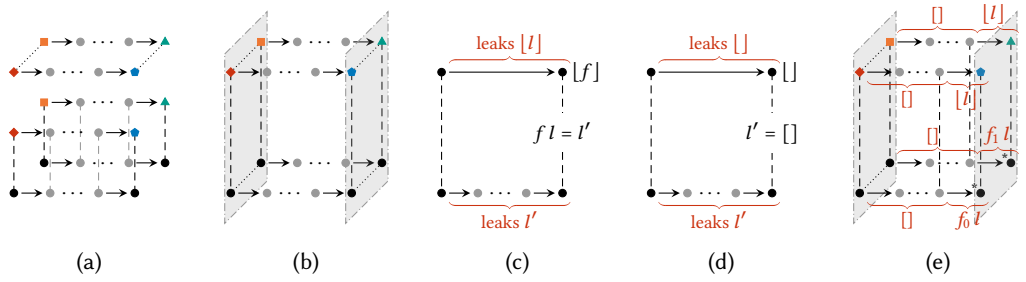


Fig. 4. (a) and (b) show how simulations can be *combined* if they have a trace in common: The nodes with the same colour and denote *identical* states. We *extend* the *horizontal* simulation on top to the traces below using the *vertical* simulation. (c) and (d) show the *local leakage transformer simulation*: (c) If there is leakage l on source, then the leakage on target l' should *adhere* to the annotated transformer f. (d) If there is no leakage on source, then there neither should be any on the target. (e) shows the *indistinguishability simulation*, where a simulation on the leakage of the source traces is extended to the target traces using local leakage transformer simulations.

As there is progress on both traces in either case, we can use maxStall $x\ j = 0$ for any position x and any lane j. This definition ensures leakage equality at every synchronised position.

The second ingredient is a simulation proof between any source trace and its related target trace, that ensures adherence to the local leakage transformer annotation. This is illustrated in Figs. 4c and 4d. There are two cases: If a source state x emits some leakage l, we expect an annotation of a leakage transformer f that maps source leakage events to sequences of target leakage events (see (c)). The sequence of leakage events of this step on the target trace has to be equal to $f\ l$. If x is not leaky, the associated step in the target should not be either (see (d)).

Now, we combine both components to build the indistinguishability simulation (see Fig. 4e). The indistinguishability assumption (depicted horizontally) ensures that the source leakages are equal and that we step from one such leakage to another. The local leakage transformer simulation (depicted vertically) relates source leakages to target leakages by the local leakage transformer annotations (denoted by f_0 and f_1). If $f_0\ l = f_1\ l$ holds, we can conclude indistinguishability on the target traces. In particular, this is the case when the functions f_0 and f_1 are equivalent.

Preserving CT. The adherence to a local leakage transformer annotation is sufficient for CT-preservation: In CT, branching decisions and memory accesses are leaked. This has two consequences: First, source traces on *public*-equal input are forced to have the same control flow. Additionally, every program point either always leaks something or never. Both properties together imply that related leakage events of the indistinguishability simulation are always at the same program point. Consequently, they have the same local leakage transformer annotation and since those in turn are applied to the same source leakage, the target leakage is equal as well.

Preserving non-interference. In the general case, as discussed in Fig. 1a, the local leakage transformers of related source leakage events may not be equal. To facilitate this equality, our backtracking code transformation NIFTY *tracks* the sequence of the local leakage transformers of individual branches and *checks* them for equality. If they are equal the optimisation can go ahead, otherwise a fallback pass is used—for this particular part of the code.

The programs in Fig. 1 exemplify the three possible cases for a conditional: In program (b) the branching predicate is *public*[8], thus any two traces with *public*-equal inputs take the same path.

[8]We say a branching predicate is *secret* if it depends on *secret* inputs and *public* otherwise.

Here, we can optimise aggressively In the programs (a) and (c), the branching variable is *secret*, thus dead code elimination might break non-interference. In (c) the first leaky assignment is deleted on both branches and the second one is kept on both branches, so non-interference is not violated and we can keep the result. However, as discussed before, the sequences of local leakage transformers in case (a) are inconsistent: $[\texttt{rm}, \texttt{id}] \neq [\texttt{id}, \texttt{rm}]$. Thus, we rewind and use a fallback pass that leaves every leaky assignment intact. The rewinding only affects the code inside of the branch.

Loops with a *public* predicate in a *public* context can be optimised aggressively as well, due to the fact that traces with *public*-equal inputs will have the same number of iterations. For loops with a *secret* predicate or that are in *secret* context, we have to use the fallback pass by default, as the track-and-check approach does not work there.

NIFTY is implemented as a higher-order function that takes a leakage model and an optimisation and its accompanying data flow transformers as inputs. The only proof obligation is a proof that target traces adhere to the local leakage transformer annotation. By design, each branch can only be backtracked once, leading to a worst-case runtime of $\mathcal{O}(f(n) + n \cdot d)$ where d is the depth of deepest nested secret branch, n the size of the program and f the runtime function of the applied optimisation.

3 Generalising Simulations to Hyperproperty Simulations

In this section we build up on the intuition given before and formalise our new notions of simulations. All of this section is independent of leakage model and the source and target languages.

3.1 k-Simulations and Synchronisers

First, we recall the standard notion of a stuttering simulation. To this end, we fix a type of source states $\mathrm{St}_{\mathrm{src}}$ and a type of target states $\mathrm{St}_{\mathrm{tgt}}$ and step functions $\mathrm{step}_{\mathrm{src}} : \mathrm{St}_{\mathrm{src}} \to \mathrm{St}_{\mathrm{src}}$ and $\mathrm{step}_{\mathrm{tgt}} : \mathrm{St}_{\mathrm{tgt}} \to \mathrm{St}_{\mathrm{tgt}}$. We introduce shared notations using $\bullet$ as placeholder. For two states x, y we write $x \longrightarrow_\bullet y$ to denote $\mathrm{step}_\bullet\ x = y$. We write $\mathrm{step}_\bullet^n\ x$ and $x \longrightarrow_\bullet^n y$ for n ($\in \mathbb{N}$) iterated applications of $\mathrm{step}_\bullet$.

At this point we do not want to fix a language and its concrete semantics. Instead, we fix an input type I and program types $\mathrm{Prog}_{\mathrm{src}}, \mathrm{Prog}_{\mathrm{tgt}}$ and assume functions $\langle \cdot, \cdot \rangle_\bullet : \mathrm{Prog}_\bullet \to \mathrm{I} \to \mathrm{St}_\bullet$ to project a program and its input to an initial state. For states x, y and a list of states a we write $x \xrightarrow{a}_\bullet^* y$ if the elements of a form a sequence of steps from x to y (including x, excluding y). The last state is not needed we just write $x \xrightarrow{a}_\bullet^*$. Note that the *totality* of $\mathrm{step}_\bullet$ implies that every pair of a program p and an input i gives rise to an *infinite* sequence of states, called the *trace of p and i*. We say a is a *trace prefix* for p and i if $\langle p, i \rangle_\bullet \xrightarrow{a}_\bullet^*$ holds.

Definition 3.1 (Stuttering Simulation). *A relation* $\approx : \mathrm{St}_{\mathrm{src}} \to \mathrm{St}_{\mathrm{tgt}} \to \mathbb{P}$[9] *is a* stuttering simulation *with respect to the two functions* $\mathrm{syncStepSt} : \mathrm{St}_{\mathrm{src}} \to \mathbb{N}$ *and* $\mathrm{maxStallSt} : \mathrm{St}_{\mathrm{src}} \to \mathbb{N}$ *on programs* $p : \mathrm{Prog}_{\mathrm{src}}$ *and* $\pi : \mathrm{Prog}_{\mathrm{tgt}}$ *and input* $i : \mathrm{I}$ *if*

- $\langle p, i \rangle_{\mathrm{src}} \approx \langle \pi, i \rangle_{\mathrm{tgt}}$ *holds, and*
- *for any source step* $x \longrightarrow y$ *and any target states* ξ, v *such that* $x \approx \xi$*, two conditions are met:*
  - *if* $\xi \longrightarrow^{\mathrm{syncStepSt}\ x} v$*, then* $y \approx v$*,*
  - *if* $\mathrm{syncStepSt}\ x = 0$*, then* $\mathrm{maxStallSt}\ x > \mathrm{maxStallSt}\ y$*.*

The necessity of the last requirement follows the same intuition as was outlined in Section 2. We call the special case where syncStepSt is constantly 1 a *lockstep simulation*. In this case no maxStallSt function has to be specified, as the last condition is trivially fulfilled.

[9] $\mathbb{P}$ denotes the type of propositions.

Our generalisation to k lanes takes a similar form: As outlined in Section 2, we require a function $\text{syncStep} : \mathbb{N}^k \to \mathbb{N}^k$ that describes the size of the *synchronised step* and a function $\text{maxStall} : \mathbb{N}^k \to \mathbb{N}_{<k}\mathbb{N}^k$ that is used to ensure eventual progress on every lane. To formulate the condition for eventual progress, we first need some auxiliary definitions. We define an iterated version of syncStep that takes a natural number $n \in \mathbb{N}$ and a position $\mathbf{m} \in \mathbb{N}^k$ and returns the *position* after n synchronised steps on $\mathbf{m}$ (where "+" is interpreted component-wise):

$$\begin{aligned} \text{syncStep}^\bullet &: \mathbb{N} \to \mathbb{N}^k \to \mathbb{N}^k \\ \text{syncStep}^0\, \mathbf{m} &= \mathbf{m} \\ \text{syncStep}^{n+1}\, \mathbf{m} &= \text{syncStep}^n\, (\mathbf{m} + \text{syncStep}\, \mathbf{m})\,. \end{aligned}$$

We introduce the notion of *synchronised* positions—the positions that are reachable by synchronised steps.

Definition 3.2. *We say* $\mathbf{n} \in \mathbb{N}^k$ *is* synchronised *with respect to* syncStep *and we write* $\text{synced}\, \mathbf{n}$ *if*

- $\mathbf{n} = 0^k$, *or*
- *there is a synchronised position* $\mathbf{m} \in \mathbb{N}^k$ *such that* $\mathbf{n} = \mathbf{m} + \text{syncStep}\, \mathbf{m}$.

Using these definitions we formalise the relationship between syncStep and maxStall that guarantees eventual progress on all lanes, a property which is commonly called *fairness*.

Definition 3.3. *The functions* (syncStep, maxStall) *form a* (k-)synchroniser *if for any synchronised position* $n \in \mathbb{N}^k$ *and any lane* j *we have*

$$n_j < (\text{syncStep}^{1+\text{maxStall}\, n\, j}\, n)_j \qquad \text{(fairness)}.$$

The strict inequality in Definition 3.3 implies that there is always eventually progress on all lanes and thus every possible position is caught up eventually.

Properties on k-tuples of traces (for a fixed $k \in \mathbb{N}$) are commonly called *hyperproperties*. We follow this convention and analogously use the term *hypertrace* for k-tuples of traces as well as *hyperstate* for k-tuples of states. Additionally, we use the term *hypersequence* to refer to a k-tuple of (finite) lists of states. Synchronisers allow us to define steps from one synchronised position on a hypertrace to the next. We call these steps *synchronised steps*. We call the components of hypertraces *lanes* and we use subscripts $\cdot_j$ to access the jth entry on any kind of tuple. We overload the $\longrightarrow$ notation in all its variants for hyperstates and hypersequences and we overload $\langle \cdot, \cdot \rangle_\bullet$ for tuples of programs and inputs. Furthermore, we define $|\cdot| : [\text{St}_\bullet]^k \to \mathbb{N}^k$ to compute the component-wise length of the respective hypersequences.

Definition 3.4 (Synchronised Step). *Assume a* k*-synchroniser* $S = (\text{syncStep}, \text{maxStall})$. *Let* p *be a* k*-tuple of programs, let* i *be a* k*-tuple of inputs and* $a : [\text{St}_\bullet]^k$ *be a hypersequence. We write* $\langle p, i \rangle_\bullet \xrightarrow{a}{}^*_S$ *if* $\text{synced}_S\, |a|$ *holds and we have* $\langle p_j, i_j \rangle_\bullet \xrightarrow{a_j}{}^*_\bullet$ *for every lane* j. *In this case, we say* a *is a* hypertrace prefix *for* p *and* i. *We write* $\langle p, i \rangle_\bullet \xrightarrow{a}{}^*_S \xrightarrow{b}_S x$ *to additionally require for every lane* j, *that* $y_j \xrightarrow{b_j}{}^{(\text{syncStep}\, |a|)_j}_\bullet$ *holds, where* y *is the last element of* a.

We define k-simulations to preserve a simulation relation over synchronised steps:

Definition 3.5. *Assume a tuple of programs* p, *a tuple of inputs* i *and let* (syncStep, maxStall) *be a* k*-synchroniser.*

A relation $\approx : [\text{St}]^k \to \mathbb{P}$ *is a* k*-simulation on* p *and* i *if the following conditions hold:*

$$\begin{aligned} &\approx \langle p, i \rangle_\bullet && \text{(base)} \\ \forall a, b : [\text{St}_\bullet]^k.\ &\approx a \Rightarrow \langle p, i \rangle_\bullet \xrightarrow{a}{}^*_S \xrightarrow{b}_S \Rightarrow \approx ab && \text{(synchronised step)} \end{aligned}$$

This definition ensures that the relation holds at every synchronised position. The defining property of the synchroniser ensures that for any hypertrace prefix a there is an extension b such that ab[10] is at a synchronised position.

Theorem 3.1 (Soundness of k-simulation). *Let a be a hypertrace prefix for p and i. If $\approx$ is a k-simulation on input i, then there is an extension b of a such that*

$$\langle p, i\rangle_\bullet \xrightarrow{ab}{}^*_\bullet \text{ and } \approx(ab).$$

Proof sketch. Since the synchroniser guarantees eventual progress on every lane, there is an $n \in \mathbb{N}$ such that $|a| \leq \text{syncStep}^n\, 0^k$[11]. We can construct a hypertrace prefix a' for p and i with $|a'| = \text{syncStep}^n\, 0^k$ and then a must be a prefix of a' as $\text{step}_\bullet$ is deterministic. Since $\approx$ is a k-simulation and $|a'|$ is a synchronised position we have $\approx a'$. □

Definition 3.6. *Assume a stuttering simulation* $(\approx, \text{syncStepSt}, \text{maxStallSt})$ *on a source program p, a target program π and an input i. We define a* 2*-simulation* $(\approx', \text{syncStep}, \text{maxStall})$ *by the following equations:*

$$\begin{aligned}
a \approx' \alpha &:= \text{last}\, a \approx \text{last}\, \alpha \qquad \textit{where last denotes the last element of a list}\\
\text{syncStep}\,(n, m) &:= (1, \text{syncStepSt}(\text{step}^n_{\text{src}}\langle p, i\rangle_{\text{src}}))\\
\text{maxStall}\,(n, m)\, 0 &:= 1\\
\text{maxStall}\,(n, m)\, 1 &:= \text{maxStallSt}(\text{step}^n_{\text{src}}\langle p, i\rangle_{\text{src}}).
\end{aligned}$$

Lemma 3.2. *Definition 3.6 indeed defines a k-simulation.*

Proof Sketch. By applying syncStepSt and maxStallSt to $\text{step}^n_{\text{src}}\langle p, i\rangle_{\text{src}}$ we are essentially translating a state-wise definition to a trace-wise one and can use progress guarantee of the former for the latter. $\approx'$ is a 2-simulation because the definition of syncStep mimics the steps on the stuttering simulation. □

Stuttering simulations between source and target traces are *total* on the reachable source states, i.e. they relate every reachable source state with a target state, which is not in general the case for synchronisers. As we will need it for combining k-simulations, we formalise totality for synchronisers.

Definition 3.7. *We say a* 2*-synchroniser* $(\text{syncStep}, \text{maxStall})$ *is* total *if there is a function* $\text{syncStep}^\sharp : \mathbb{N} \to \mathbb{N}$ *such that for any $n \in \mathbb{N}$ we have*

$$\text{synced}(n, \text{syncStep}^\sharp\, n).$$

Lemma 3.3. *The synchroniser in Definition 3.6 is total.*

Proof Sketch. By using the fact that syncStep always steps 1 step on source. □

We will now investigate k-simulations that are specifically suited for preserving hyperproperties.

[10] For sequences or hypersequences a, b we write to ab to denote their concatenation.
[11] Where $\leq$ is defined component-wise.

3.2 Hyperproperty Simulations

The preservation (or transformation) of a k-hyperproperty P is basically a $2k$-hyperproperty: the conjunction of having P on the source and on the target traces. In this section we introduce *hyperproperty simulations* which are instances of k-simulations and thus share their soundness results. As seen in the previous section, a k-simulation needs some sort of synchronisation. We define the synchroniser *modularly* by combining a k-synchroniser on the source traces with the 2-synchroniser between source and target traces induced by a stuttering simulation. This *combined* synchroniser is constructed such that every synchronised position is a synchronised position with respect to the source synchroniser as well as to the source-target synchroniser (and thus source and target states are related by the simulation relation of the stuttering simulation). This approach allows for different recombinations of synchronisers.

To be able to formulate hyperproperty simulations we first construct the combined synchroniser. Let p be a source program, π be a target program and let i be a k-tuple of inputs. We write $\langle (p,i),(\pi,i)\rangle_{2k}$ to denote the initial $2k$-hyperstate that is induced by the cartesian product of (p,π) and i. This initial hyperstate induces a $2k$-hypertrace. By convention we call the first k lanes the source lanes and the second k lanes the target lanes. We assume a k-synchroniser $(\mathrm{syncStep}_H, \mathrm{maxStall}_H)$ on the source lanes and call it the *source synchroniser*. In Fig. 4a it is represented by the *horizontal* component. Additionally, we assume a family of source-target synchronisers $(\mathrm{syncStep}_{V_j}, \mathrm{maxStall}_{V_j})_{j\in\mathbb{N}_{<k}}$ consisting of k total 2-synchronisers that relate lane j and $j+k$, for $0 \le j < k$. Given a stuttering simulation on any input, this family can be constructed using Definition 3.6. The source-target synchronisers are represented by the *vertical* component in Fig. 4a.

As discussed in Section 2, we extend the source synchroniser to the target lanes by using the projection of the source-target synchronisers to retrieve the corresponding target position. We define $\mathrm{syncStep}_{2k}$ for the combined synchroniser. For $\mathbf{n},\mathbf{m} \in \mathbb{N}^k$, let

$$\mathrm{syncStep}_{2k}(\mathbf{n},\mathbf{m}) = \left(\mathrm{syncStep}_H \mathbf{n}, \left(\mathrm{syncStep}_{V_j}^\sharp\left((\mathbf{n} + \mathrm{syncStep}_H \mathbf{n})_j\right)\right)_{j\in\mathbb{N}_{<k}} - \mathbf{m}\right).$$

Note that we add $\mathbf{n}$ and subtract $\mathbf{m}$ in this formula because $\mathrm{syncStep}_H$/$\mathrm{syncStep}_{2k}$ return the *step size* while $\mathrm{syncStep}_{V_j}^\sharp$ assumes and returns an absolute position. For the source lanes $\mathrm{maxStall}_{2k}$ is just defined by $\mathrm{maxStall}_H$:

$$\mathrm{maxStall}_{2k}\ 2k(\mathbf{n},\mathbf{m})\ j = \mathrm{maxStall}_H \mathbf{n}\ j \text{ where } 0 \le j < k.$$

We omit the definition of $\mathrm{maxStall}_{2k}$ for the target lanes because it is quite technical and not very insightful. The intuition is that since $\mathrm{maxStall}_H$ guarantees progress on the source lanes and $\mathrm{maxStall}_{V_j}$ guarantees progress on the target lanes, if there is progress on the source lanes, we can use $\mathrm{maxStall}_{V_j}$ to figure out how often $\mathrm{maxStall}_H$ has to be applied to progress on the target lanes. This construction ensures progress on any lane for the combined synchroniser.

Theorem 3.4. *The functions* $(\mathrm{syncStep}_{2k}, \mathrm{maxStall}_{2k})$ *form a* $2k$*-synchroniser called the* combined *synchroniser. For any* $\mathbf{n} \in \mathbb{N}^{2k}$ *with* $\mathrm{synced}_{2k}\,\mathbf{n}$*, corresponding horizontal* ($\mathbf{m} \in \mathbb{N}^k$) *and vertical* ($\mathbf{m} \in \mathbb{N}^2$) *positions are synchronised positions with respect to* $\mathrm{syncStep}_H$ *and* $\mathrm{syncStep}_{V_j}$*, respectively.*

As a notational convenience we define

$$\langle (p,i),(\pi,i)\rangle_{2k} \xrightarrow{(a,\alpha)}{}^{*}_{V} \xrightarrow{(b,\beta)}{}^{*}_{V} := \forall 0 \le j < k, \langle (p_j,i_j),(\pi_j,i_j)\rangle_{2k} \xrightarrow{(a_j,\alpha_j)}{}^{*}_{V_j} \xrightarrow{(b_j,\beta_j)}{}^{*}_{V_j}.$$

Definition 3.8. *Let* $\approx_{\mathrm{src}} : [\mathrm{St}_{\mathrm{src}}]^k \to \mathbb{P}$ *and* $\approx_{\mathrm{tgt}} : [\mathrm{St}_{\mathrm{tgt}}]^k \to \mathbb{P}$ *be relations.*

We say $(\approx_{\text{src}}, \approx_{\text{tgt}})$ *is a* hyperproperty simulation *if the following conditions hold:*

$$\begin{aligned}
&\underset{\text{src}}{\approx} \langle p, i\rangle_{\text{src}} && \text{(base source)}\\
&\underset{\text{tgt}}{\approx} \langle \pi, i\rangle_{\text{tgt}} && \text{(base target)}\\
&\forall a\, \alpha,\ \underset{\text{src}}{\approx} a \Rightarrow \underset{\text{tgt}}{\approx} \alpha \Rightarrow && \\
&\quad \forall b\, \beta,\ \langle p, i\rangle_{\text{src}} \xrightarrow{a} {}^{*}_{H} \xrightarrow{b} {}_{H} && \\
&\qquad \Rightarrow \langle (p,i),(\pi,i)\rangle_{2k} \xrightarrow{(a,\alpha)} {}^{*}_{V} \xrightarrow{(b,\beta)} {}^{*}_{V} && \text{(hyperproperty step)}\\
&\qquad \Rightarrow \underset{\text{src}}{\approx}(ab) \wedge \underset{\text{tgt}}{\approx}(\alpha\beta). &&
\end{aligned}$$

This definition is very similar to Definition 3.5: Essentially we require a synchronised step on the source synchroniser as well as a *sequence* of synchronised steps on every source-target synchroniser. The reason for this subtle difference is that the combined synchroniser does guarantee a synchronised position for the source-target synchroniser, but multiple steps may be required to get there.

Theorem 3.5 ($2k$-simulation). *A k-hyperproperty simulation $(\approx_{\text{src}}, \approx_{\text{tgt}})$ is a $2k$-simulation with $\approx(a,\alpha) := \approx_{\text{src}} a \wedge \approx_{\text{tgt}} \alpha$.*

Proof. The base case follows directly from the hyperproperty simulation base cases. For the synchronised step we have to show $\approx(ab, \alpha\beta)$, given $\approx(a,\alpha)$ and $\langle (p,i),(\pi,i)\rangle_{2k} \xrightarrow{a} {}^{*}_{B} \xrightarrow{b} {}_{B}$. Using Theorem 3.4 we can derive

$$\langle p, i\rangle_{\text{src}} \xrightarrow{a} {}^{*}_{H} \xrightarrow{b} {}_{H} \text{ and } \langle (p,i),(\pi,i)\rangle_{2k} \xrightarrow{(a,\alpha)} {}^{*}_{V} \xrightarrow{(b,\beta)} {}^{*}_{V}$$

from the latter assumption. Now, applying the hyperproperty step concludes the proof. □

Theorem 3.6 (Soundness of Hyperproperty Simulation). *Let a and α be hypertraces prefixes for i and p or π, respectively. If $(\approx_{\text{src}}, \approx_{\text{tgt}})$ is a hyperproperty simulation then there are extensions b and β, such that*

$$\underset{\text{src}}{\approx}(ab) \wedge \underset{\text{tgt}}{\approx}(\alpha\beta) \wedge \langle p, i\rangle_{\text{src}} \xrightarrow{ab}{}^{*}_{\text{src}} \wedge \langle \pi, i\rangle_{\text{tgt}} \xrightarrow{\alpha\beta}{}^{*}_{\text{tgt}}.$$

Proof. Use Theorem 3.5 and Theorem 3.1. □

3.3 Indistinguishability Simulations

To ease proofs of preserving indistinguishability properties like non-interference, we design a specific hyperproperty simulation instance. This allows us to modularise the proof into a *horizontal* and a *vertical* component. The horizontal component consists of the source leakage equivalence and the equality of local leakage transformers between two related leakage events. The vertical component is the adherence of the target leakage to the local leakage transformers, i.e. a simulation argument between a source and target trace stating that the target traces indeed produce the leakage as it is described by the annotated local leakage transformers.

We model leakage events with a type L that represents what can be observed and a *leakage function* $\mathcal{L}$. The leakage function maps states to $\lfloor L \rfloor$ where $\lfloor \cdot \rfloor$ denotes the option type constructor, i.e. something of type $\lfloor L \rfloor$ is either $\lfloor\rfloor$ (no leakage) or $\lfloor x \rfloor$ where x is of type L. We define a state x to be *leaky* if $\mathcal{L}\ x \neq \lfloor\rfloor$ and a hyperstate to be leaky if x_j is leaky on *all* lanes j. We lift $\mathcal{L}$ to sequences in the following way (where $a :: x$ denotes appending an element x to the list a):

Definition 3.9 (Leakage Sequence). *For $\mathcal{L} : \mathrm{St} \to \lfloor L \rfloor$ we define $\overline{\mathcal{L}} : [\mathrm{St}] \to [L]$:*

$$\overline{\mathcal{L}}\,[\,] = [\,]$$
$$\overline{\mathcal{L}}\,(a :: x) = \begin{cases} \overline{\mathcal{L}}\,a & \text{if } \mathcal{L}\,x = \lfloor\,\rfloor \\ \overline{\mathcal{L}}\,a :: l & \text{if } \mathcal{L}\,x = \lfloor l \rfloor\,. \end{cases}$$

We say a sequence of states is *leak-free* if $\overline{\mathcal{L}}\ x = [\,]$ holds. A hypersequence is leak-free if for every lane the respective list is leak-free. Note that the terms *leaky* and *leak-free* are defined on different domains and are no complements to each other.

Horizontal Component. For the remainder of this section we fix a leakage function $\mathcal{L}$ and an equivalence relation $\equiv$ on lists of leakage events. Now, we can define *indistinguishability* on trace prefixes and *eventual indistinguishability* (capturing a property coined "coproductive security" in [Bohannon et al. 2009]) and *non-interference* of programs in our setup.

Definition 3.10 (Indistinguishability). *We say two lists of states a, a' are* indistinguishable *and write $a \equiv_{\mathcal{L}} a'$ if $\overline{\mathcal{L}}\ a \equiv \overline{\mathcal{L}}\ a'$ holds.*

Definition 3.11 (Eventual Indistinguishability). *We say initial states $\langle p, i\rangle_\bullet$ and $\langle p', i'\rangle_\bullet$ observe* eventual indistinguishability *if for any two sequences a, a' such that $\langle p, i\rangle_\bullet \xrightarrow{a}{}^*_\bullet$ and $\langle p', i'\rangle_\bullet \xrightarrow{a'}{}^*_\bullet$ there are extensions b, b' such that $\langle p, i\rangle_\bullet \xrightarrow{a}{}^*_\bullet \xrightarrow{b}{}^*_\bullet$, $\langle p', i'\rangle_\bullet \xrightarrow{a'}{}^*_\bullet \xrightarrow{b'}{}^*_\bullet$ and $ab \equiv_{\mathcal{L}} a'b'$.*

Definition 3.12 (Non-Interference). *Let $\sim$ be an equivalence relation on inputs (denoting equality of* public *inputs). We say a program observes* non-interference *if for any two inputs i, i' such that $i \sim i'$, the initial states $\langle p, i\rangle_\bullet$ and $\langle p, i'\rangle_\bullet$ observe eventual indistinguishability.*

When preserving non-interference we want to use the assumption of non-interference on the source program to instantiate a 2-simulation on the source traces that synchronises on leakage events. As k-simulations are defined on infinite traces and always require eventual progress, we have to handle the case where there is no leakage left in a special way.

Definition 3.13 (Next Leaky Position). *We define the* step to the next leaky position *for a program p, an input i and an $n \in \mathbb{N}$ as the smallest $m \in \mathbb{N}_{>0}$ such that there is a state x such that $\langle p, i\rangle_\bullet \longrightarrow^n_\bullet \longrightarrow^m_\bullet x$ holds and x is leaky (i.e. $\mathcal{L}\,x \neq \lfloor\,\rfloor$).*

We say $\mathbf{m} \in \mathbb{N}^k_{>0}$ is the hyperstep to the next leaky position *for a program p, an input tuple i and a position $\mathbf{n} \in \mathbb{N}^k$, if $\mathbf{m}_j$ is the next leaky position on p, i_j and $\mathbf{n}_j$ for all lanes j.*

The existence of a next leaky position is not in general decidable, but since $\mathrm{syncStep}_H$ does not have to be computable we can exploit the axiom of choice and use a classical case distinction to define the *leakage synchroniser*.

Definition 3.14 (Leakage Synchroniser). *For a program p and an input tuple i we define a* leakage synchroniser*:*

$$\mathrm{syncStep}_H\,\mathbf{n} = \begin{cases} \mathbf{m} & \text{if } \mathbf{m} \in \mathbb{N}^k_{>0} \text{ is the step to a next leaky position on } p,\ i \text{ and } \mathbf{n} \\ 1^k & \text{if no such next leaky position exists} \end{cases}$$
$$\mathrm{maxStall}_H\,\mathbf{n}\ j = 1.$$

Lemma 3.7. *The* leakage synchroniser *is indeed a synchroniser.*

Proof. On every synchronised step the synchroniser makes progress on all traces: If there is a next leaky position, then by the requirement of $m > 0$ in Definition 3.13. Otherwise by definition. □

Lemma 3.8. *Let p be a program and i, i' be inputs. The initial states $\langle p, i\rangle_\bullet$ and $\langle p, i'\rangle_\bullet$ observe eventual indistinguishability if and only if $\equiv_{\mathcal{L}}$ is a 2-simulation on $\langle p, i\rangle_\bullet$ and $\langle p, i'\rangle_\bullet$ (synchronised by the leakage synchroniser).*

Proof. In the forward direction, use the extension given by Definition 3.11 to construct the synchronised step for the 2-simulation. In the other direction: By Theorem 3.1 for any trace prefixes a, a' there are semantically valid extensions b, b' such that $ab \equiv_{\mathcal{L}} a'b'$. □

Vertical Component. To relate source and target leakage events, we now assume a type L' for target leakage events. Additionally we assume a function $\mathrm{TF} : \mathrm{St}_{\mathrm{src}} \to \lfloor L \to [L'] \rfloor$ that maps every source state to an option type of a *local leakage transformer*.

Definition 3.15 (Adherence of Local Leakage Transformer). *Assume a source sequence of states a and a target sequence of states α. We say the leakage on α adheres to the local leakage transformers of a and write $a \approx_V \alpha$ if either*

- *both lists are empty, i.e. $a = [\,] = \alpha$*
- *$a = a' :: x$ and $\alpha = \alpha'\beta$ for a state x and lists of states a', α', β such that $a' \approx_V \alpha'$ and*
  - *if $\mathcal{L}\, x = \lfloor\,\rfloor$, then $\overline{\mathcal{L}}\,\beta = [\,]$*
  - *if $\mathcal{L}\, x = \lfloor l \rfloor$, then $\overline{\mathcal{L}}\,\beta = \mathrm{TF}\, x\, (\mathcal{L}\, x)$.*

Definition 3.16 (Local Leakage Transformer Simulation). *We say we have a* local leakage transformer simulation *on programs p, π and input i if $\approx_V$ is a total 2-simulation on $\langle p, i\rangle_{\mathrm{src}}$ and $\langle \pi, i\rangle_{\mathrm{tgt}}$.*

Definition 3.17 (Horizontal Consistency of Local Leakage Transformers). *We say we have* horizontal consistency of local leakage transformers *on p and k-tuple of inputs i, if $\approx_H$ is a k-simulation on $\langle p, i\rangle_{\mathrm{src}}$, where $\approx_H$ is defined as the equality of* TF *on the last states of the respective trace prefixes.*

Now, we have everything to state and prove our main theorem.

Theorem 3.9 (Indistinguishability Simulation). *Let p be a source program and π be a target program and let i_0, i_1 be inputs. If*

- *p,i_0,i_1 observe horizontal consistency of local leakage transformers,*
- *$\approx_V$ is a local leakage transformer simulation on both inputs, and*
- *$\equiv_{\mathcal{L}}$ is a 2-simulation on $\langle p, i_0\rangle_{\mathrm{src}}$ and $\langle p, i_1\rangle_{\mathrm{src}}$,*

then $(\approx_H, \approx_H)$ is a hyperproperty simulation on $\langle (p, (i_0, i_1)), (\pi, (i_0, i_1))\rangle_{2k}$. We call this instance an indistinguishability simulation.

Proof. The base cases hold trivially. The horizontal synchroniser is given by the leakage simulation and the vertical synchronisers are given by the local leakage transformer simulations. It remains to show the hyperproperty step of Definition 3.8 holds:

Assume $a : {\mathrm{St}_{\mathrm{src}}}^2$ and $\alpha : {\mathrm{St}_{\mathrm{tgt}}}^2$ such that $a_0 \equiv_{\mathcal{L}} a_1$ and $\alpha_0 \equiv_{\mathcal{L}} \alpha_1$ hold. Furthermore, we have b, β, x such that $\langle p, i\rangle_{\mathrm{src}} \xrightarrow{a}{}^{*}_{H} \xrightarrow{b::x}_{H}$ and $\langle (p, i), (\pi, i)\rangle_{2k} \xrightarrow{(a,\alpha)}{}^{*}_{V} \xrightarrow{(b::x,\beta)}{}^{*}_{V}$ hold, where the deconstruction of the synchronised step into $b :: x$ is justified by the fact that the leakage synchroniser always steps forward on all lanes. Now, we do a case distinction on whether x is leaky:

- If x_0 is leaky but x_1 is not or vice versa, we have a contradiction to the indistinguishability assumption.

- If x_0 and x_1 are leaky, we can derive the following equation:
$$\overline{\mathcal{L}}\,\beta_0 \equiv \mathrm{TF}\,x_0\,(\mathcal{L}\,x_0) \equiv \mathrm{TF}\,x_1\,(\mathcal{L}\,x_0) \equiv \mathrm{TF}\,x_1\,(\mathcal{L}\,x_1) \equiv \overline{\mathcal{L}}\,\beta_1.$$
The first and last step are justified by the local leakage transformer simulation, the second step is justified by the horizontal consistency of local leakage transformers and the third step is justified by indistinguishability on the source.
- If neither x_0 nor x_1 are leaky, we have $\overline{\mathcal{L}}\,\beta_0 = [\,] = \overline{\mathcal{L}}\,\beta_1$ by the local leakage transformer simulations.

Thus, we have $\alpha_0\beta_0 \equiv_{\mathcal{L}} \alpha_1\beta_1$ and by the indistinguishability assumption also $a_0 b_0 \equiv_{\mathcal{L}} a_1 b_1$. □

Corollary 3.10. *Let p be a non-interfering source program and let π be a target program. If conditions (1) and (2) of Theorem 3.9 hold for any pair of* public*-equal inputs, π is non-interfering.*

Proof. Use Theorem 3.8 to derive the third condition and to conclude the proof. □

4 Non-Interference Preserving Optimisations

In this section we demonstrate important applications of Theorem 3.10. First, we use it to simplify CT-preservation proofs. Then, we introduce our code transformation framework NIFTY to support leakage models that allow *secret* branches. NIFTY uses backtracking to ensure horizontal consistency of local leakage transformers (see Theorem 3.9), it thereby adjusts optimisations to preserve non-interference even in the presence of *secret* branches. Finally, we instantiate both concepts on some example optimisations.

4.1 Preserving Constant-Time

As has been argued in Section 2, we can reduce the proof burden of CT-preservation to simply proving the target leakage to adhere to the local leakage transformer annotation. This is due to the fact that in CT any two related leakages must occur at the same program point.

$$\begin{aligned} p \in \mathrm{Prog} &:= \mathrm{list}\,\mathrm{Stmt} \\ \mathrm{Stmt} &:= x \leftarrow e \;\#\mathrm{A} \\ &\mid \texttt{if}\,e\,\texttt{then}\,p_0\,\texttt{else}\,p_1 \;\#\mathrm{A} \\ &\mid \texttt{while}\,e\,\texttt{do}\,p_0 \;\#\mathrm{A} \end{aligned}$$

where e is an expression and x is variable.

(a) The syntax of our minimal language.

$$\frac{}{\langle x \leftarrow e, i\rangle; p \longrightarrow \langle p, \{i\{x \mapsto [e]_i\}\rangle} \qquad \frac{[e]_i \neq 0}{\langle \texttt{if}\,e\,\texttt{then}\,p_1\,\texttt{else}\,p_2; p, i\rangle \longrightarrow \langle p_1; J; p, i\rangle}$$

$$\frac{}{\langle J; p, i\rangle \longrightarrow \langle p, i\rangle} \qquad \frac{}{\langle [\,], i\rangle \longrightarrow \langle [\,], i\rangle}$$

(b) An excerpt of the semantics of our minimal language.

Fig. 5

To formalise this, we first define a minimal language and its semantics. Our language consists of three kinds of *statements*: assignments, conditionals and loops. A *program* is a list of statements. Fig. 5a presents the syntax. We assume a set of variables Var and define the types Val := $\mathbb{N}$, Env := Var → Val. Instead of specifying expressions we assume a type of expressions Expr and a evaluation function $[\cdot]$. as well as a function mem returning all contained variables. Expressions are *pure*, they have no external effects. Our syntax supports annotations at statements: We write $s\#A$ to annotate the statement s with some additional information A. We will omit the annotations when they are not relevant.

The semantics are mostly as one would expect them, Fig. 5b shows an excerpt. The *program state* $\langle p, i\rangle$ consists of a remaining program p and a *variable environment* i: A step in the semantics processes the head of p and adjusts the remaining program and variable environment of the next program state accordingly. For example, the execution of an assignment $x \leftarrow e$ just removes it from the remaining program and sets x to the value of the expression e in the variable environment of the next state. To be able to distinguish code in branches from the remaining program, we introduce a special no-op statement J when evaluating conditionals and loops. As we require the step relation in Section 3 to be a function, we stutter the termination state. We use variable environments i as input type, thus the initial state of a program p and an input i is just $\langle p, i\rangle$.

To model leakage in our semantics we assume a type L and a synctactic leakage function

$$\ell : \mathrm{Stmt} \to \lfloor \mathrm{Env} \to L \rfloor .$$

The leakage function takes a statement as input and returns an option type of a function. This guarantees that the decision of whether there is leakage is statically available, but not necessarily the specific value of it. We instrument our semantics with the leakage function:

$$\langle p, i\rangle; l \longrightarrow_\ell \langle p', i'\rangle; \ell\, s\, i, \text{ where } \langle p, i\rangle \longrightarrow \langle p', i'\rangle.$$

Executing a statement s results in having the leakage $\ell\, s\, i$ in the *next* instrumented state. We assume termination to be observable as well and realise this by a special termination leakage T. We instantiate the *semantic* leakage function $\mathcal{L} : \mathrm{St} \to \lfloor L \rfloor$ from Section 3 by as $\mathcal{L}\langle p, i\rangle; l := l$.

Let variables be partitioned into subsets of *public* and *secret* variables:$\mathrm{Var}_{\mathrm{Pub}} \uplus \mathrm{Var}_{\mathrm{Sec}} = \mathrm{Var}$. Now, we can define CT where we model memory accesses with accesses to *public* variables.

Definition 4.1 (Cryptographic Constant-time Leakage Model).

$$\ell_{CT}(x \leftarrow e) := \begin{cases} \lfloor \rfloor & \text{if } x \in \mathrm{Var}_{\mathrm{Sec}} \wedge \mathrm{Var}_{\mathrm{Pub}} \cap \mathrm{mem}\, e = \{\} \\ \lfloor \lambda\_.(\lfloor \rfloor, \lfloor x \rfloor, \mathrm{Var}_{\mathrm{Pub}} \cap \mathrm{mem}\, e) \rfloor & \text{if } x \in \mathrm{Var}_{\mathrm{Pub}} \\ \lfloor \lambda\_.(\lfloor \rfloor, \lfloor \rfloor, \mathrm{Var}_{\mathrm{Pub}} \cap \mathrm{mem}\, e) \rfloor & \text{otherwise} \end{cases}$$

$$\left.\begin{array}{l} \ell_{CT}(\texttt{if}\, e \ldots) \\ \ell_{CT}(\texttt{while}\, e \ldots) \end{array}\right\} := \left\lfloor \lambda i.\left(\left\lfloor [e]_i \neq^? 0 \right\rfloor, \lfloor \rfloor, \mathrm{Var}_{\mathrm{Pub}} e\right) \right\rfloor .$$

Definition 4.2. *We say a program p observes* cryptographic constant-time *(in short:* CT*) if it observes non-interference with respect the leakage model* $\mathrm{St}_{\mathrm{src}CT}$.

Lemma 4.1. *Let p be a program that observes CT. Then, any two related leakage events are at the same program point.*

Proof Sketch. This property is a core motivation for CT. We can prove it with a lockstep simulation on p and two *public*-equal inputs. At every step, as branching decisions are leaked, both traces must always take the same branch and since by definition any statement either is leaky or not—independent of the variable environment—related leakage events are always at the same program point. □

We say a program p is annotated with local leakage transformers if for any substatement the annotation provides a function $f : \lfloor L \to [L'] \rfloor$, where L' denotes the type of target leakage events.

Theorem 4.2. *Let p be a source program that is annotated with local leakage transformers and let π be a target program. We assume there is a local leakage transformer simulation (see Definition 3.16) between p and π on any input i. Then, CT on p implies CT on π.*

Proof. We apply Theorem 3.10. Conditions (2) and (3) are given by assumption, only horizontal consistency remains to be shown. By Theorem 4.1, any two related leakages must be at the same program point and since annotations *only* depend on the program point this concludes the proof. □

The proof of Theorem 4.2 does not depend on our specific definition of CT—the statement holds for any leakage model satisfying Theorem 4.1. This is the case for any reasonable formulation of CT.

4.2 Preserving NI

In the previous subsection the leakage model ensured horizontal consistency of local leakage transformers. In the presence of *secret* branches this is not the case. To guarantee horizontal consistency nonetheless, we introduce the backtracking code transformation NIFTY. As discussed before, the core idea is to optimise code inside *secret* branches aggressively, but check at every control flow rejoin for horizontal consistency of local leakage transformers.

We optimise the program in two steps: First, we annotate the source program. This annotation contains the local leakage transformers at every leaky source program point and it determines where the considered optimisation is *disabled*. In this step the backtracking technique discussed before is used to ensure horizontal consistency (see Definition 3.17) of local leakage transformers. By Theorem 3.10, it is now safe to apply the optimisation to the annotated program. The only requirement for the optimisation pass is that it can be disabled by such an annotation and that it indeed changes the leakage in the way the annotation says.

We assume optimisations to be guided by some data flow analysis. For example, in the case of dead code elimination the analysis would be a *liveness analysis* that evaluates for every program point the set of *live variables*, i.e. variables that will potentially be used after the program point and before they are overwritten. Only assignments that are *dead*, i.e. whose variable is *not live* at that point, may be eliminated. Furthermore, we need a function that maps analysis results to *local leakage transformers*.[12] As demonstrated in Section 1 these can be represented by `rm` and `id` for dead code elimination and they directly correspond to the liveness analysis result.

When the optimisation is disabled in a *secret* branch, as for example in Fig. 1a, this may affect the result of the data flow analysis even outside the branch: For the example of dead code elimination, the elimination of an assignment may shorten the lifetime of the variables it uses. To ensure sound analysis results and avoid reanalysing the whole program, we *integrate* the data flow analysis into our annotation pass for local leakage transformers.

Even when disabling the optimisation *locally* we still need sound information on the data flow of this section. For example, in Fig. 1a we need sound liveness information for the branches of the conditional to support optimisations outside of it. Thus, we need a *fallback* data flow transformer.

Finally, depending on the optimisation we need a mapping from the local data flow analysis result to a local leakage transformer and a leakage model as defined in Section 4.1.

To summarise NIFTY takes the following inputs:

- a *optimising* data flow analysis, given as a set of data flow transformers (FTa, FTi, FTw)[13],
- a *fallback* data flow analysis, given a set of data flow transformers (FTa_0, FTi_0, FTw_0),
- a mapping from analysis results to local leakage transformers (TF),
- an annotation at every branch whether it depends on *secret* inputs (*s*#secret), and
- a leakage model.

NIFTY then does *one* data flow pass through the program where it may on backtrack conditionals (but each one only once) and iterates through loops until the data flow analysis FT finds a fixpoint. The result is a program that has a *sound* analysis annotation with respect to the original data flow analysis as well as local leakage transformer annotations such that the *secret* branches always have the same sequence of local leakage transformers.

[12] More precisely, we need a representation of them that have a decidable equality.

[13] One for each syntactic construct: *a*ssignment, *i*f-statement and *w*hile-statement.

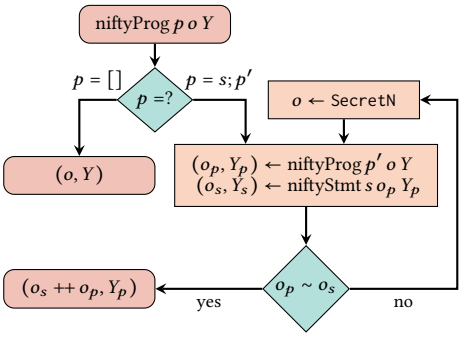


Fig. 6. A flowchart describing how niftyProg.

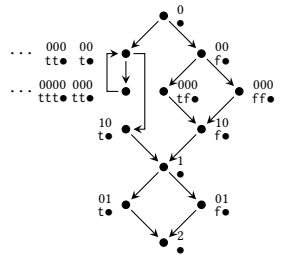


Fig. 7. An example program where every program point is annotated with its respective control flow tags. The tags on the left side of the loop describe different loop iterations.

We control the behaviour of the analysis with an *optimisation mode*. This is realised by the following inductive type:

$$\texttt{optMode} \coloneqq \texttt{Public} \mid \texttt{SecretT}\,(bl : [L \to [L']]) \mid \texttt{SecretN}.$$

The optimisation mode `Public` signifies that the control flow at this point is not dependent on secret inputs, thus any leakage here must occur on all lanes. This allows for the use of the optimising analysis. Both `SecretT` and `SecretN` signify that the control flow is dependent on *secret* inputs, i.e. different *public*-equivalent lanes may observe different occurrences of leakage. In the mode `SecretT` the algorithm keeps track of the witnessed local leakage transformers at every occurrence of leakage with a list thereof. If the tracked local leakage transformers are not equal at a join point of secret branches we use `SecretN` to disable the optimisation locally. We define $\sim$ as the predicate that takes two optimisation modes and returns whether they are of the same kind. We overload the list operations :: and ++ for optimisation modes: For optimisation modes of type `SecretT` it operates on the tracking information, otherwise both return the left operand.

NIFTY is a pass over the program that propagates both the optimisation mode as well as the analysis information to preceding or succeeding statements[14]. We implement it with the two mutual recursive functions that act on either statements or programs, respectively:

$$\texttt{niftyStmt}\,(s : \text{Stmt})\,(o : \texttt{optMode})\,(b : B) : \texttt{optMode} * B$$
$$\texttt{niftyProg}\,(p : \texttt{list}\,\text{Stmt})\,(o : \texttt{optMode})\,(b : B) : \texttt{optMode} * B.$$

Both take the optimisation mode o as well as the analysis information b of the remaining program as additional arguments and they return a pair of an optimisation mode and an analysis information. In this process niftyStmt also always directly annotates the respective program point.

Figure 6 describes how a program is processed in the backwards variant: Given an empty program, niftyProg just returns its arguments. Otherwise, it first makes a recursive call on the tail of the program and then calls niftyStmt on the head where it uses the optimisation mode and analysis information returned from the first call. If both returned optimisation modes are of the same kind, then niftyProg returns this optimisation mode with possibly appending the tracking information along with the analysis information of the call on the statement. Otherwise, the recursive calls are repeated with `SecretN` as optimisation mode. In this case it is guaranteed that the resulting optimisation modes are `SecretN` for both, thus termination is ensured.

[14]Depending on the iteration order of the original analysis it is either a forward or a backward pass.

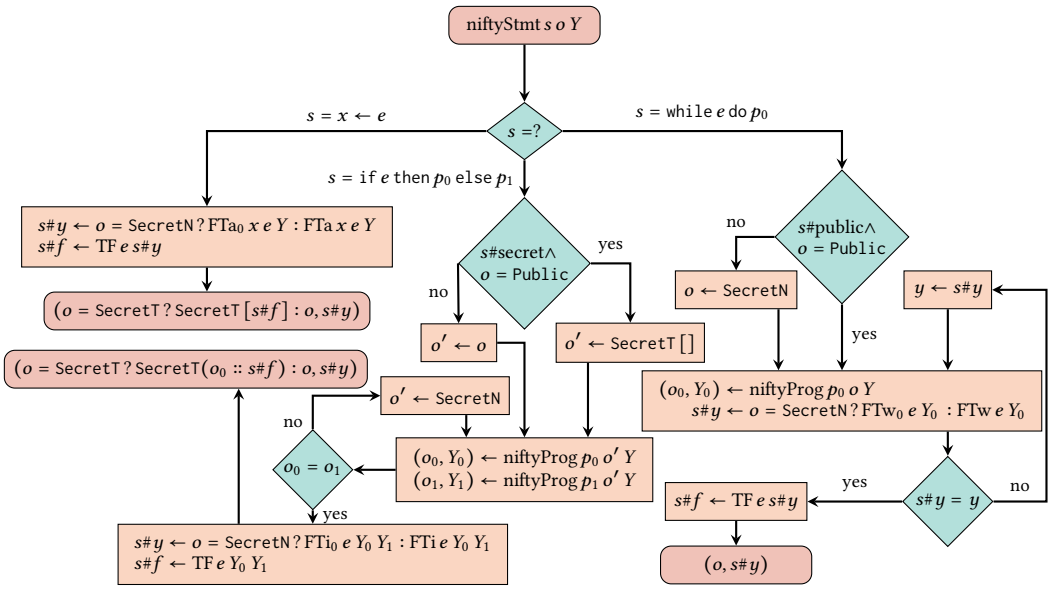


Fig. 8. The flowchart demonstrates the implementation of niftyStmt.

Now, consider Fig. 8 for a visualisation of `niftyStmt`. For an assignment we check whether the current optimisation mode is `SecretN`. If so, then the fallback analysis FT_0 is used. Otherwise, we use the standard approach. In either case we annotate the statement with the analysis result (stored in y) and the respective local leakage transformer (stored in f). The optimisation mode (possibly including updated tracking information) and the analysis result are propagated. On a conditional, we first check whether we are entering *secret* control flow: If the (outside) optimisation mode is `Public` and the branching predicate is dependent on *secret* inputs, we set the optimisation mode to `SecretT`, otherwise the outer control flow is propagated. Then, we can analyse the branches in the current optimisation mode. If these calls produce both the same optimisation mode, in particular the same tracking information (if in `SecretT`), then the result can be used and the resulting analysis result is given by the corresponding transformer for conditionals. Otherwise we have to do the analysis again with $o' = \texttt{SecretN}$.[15] If the outer control flow of a loop and its branching condition are *public* we can use the optimisation mode `Public` for its body. Otherwise, the number of loop iterations may differ on different traces which makes tracking of leakage changes impossible. Therefore we set the optimisation to `SecretN` in this case. The data flow analysis FT may need several iterations through the loop to find a fixpoint.

To ensure a semantics-preserving optimisation, NIFTY should provide the same soundness guarantees as the original analysis does.

Lemma 4.3. *Assume that the optimising data flow transformers and the fallback data flow transformers both produce* sound *annotations with respect to some soundness property of the respective optimisation. Then NIFTY produces sound annotations with respect to the same property.*

The track-and-check approach for conditionals with high branching conditions guarantees that the leaks on alternative paths through the conditional are treated the same way. Implicitly, this assumes that both paths have the same leaks in the *source* program. This is a stronger assumption than what a purely semantic definition of non-interference would provide, as here two leakage

[15]Note that this second pass necessarily succeeds because the optimisation mode will be `SecretN` on both branches.

events may be related although they are in two *different*, sequential conditionals with a *secret* predicate (e.g. `if` a `then` X `else` $\cdot$; `if` a `then` $\cdot$ `else` X, where X is some leaky event). To allow for arbitrary optimisations in *public* control flow and as many as possible in *secret* control flow we define a stronger variant of non-interference. Intuitively, we require leakage equivalence of *public*-equivalent traces at every *re-join* of control flow.

We define this requirement using *control flow tags*. Consider Fig. 7 for an example that shows how we track control flow. A control flow tag consists of two stacks of the same length, denoted on top of each other. The lower component captures the *relevant* branching decisions while the upper component captures how many control flow rejoins have already happened at any given depth. As both stacks have always the same length we can also interpret them as one stack of tuples. At the root of the program we initialise the stack with one single entry $\begin{smallmatrix}0\\\bullet\end{smallmatrix}$. Whenever the control flow branches, $\begin{smallmatrix}0\\\mathtt{t}\end{smallmatrix}$ or $\begin{smallmatrix}0\\\mathtt{f}\end{smallmatrix}$ is pushed to the stack, depending on which branch was taken. On every control flow rejoin the top of the stack is dropped and the counter of the next entry is increased. When processing a conditional or a loop the tail of the program is appended to the respective subprograms (i.e. the taken conditional branches or the loop body). Conversely, the top of the stack is popped whenever a conditional or loop is left, i.e. when witnessing the join marker J (see Section 4.1).

To make sure that leakages traces are equal at every join, we require the control flow tags of related leakage events to be *aligned*.

Definition 4.3. *Let* $\begin{smallmatrix}c_0\\d_0\end{smallmatrix}, \begin{smallmatrix}c_1\\d_1\end{smallmatrix}$ *be control flow tags. Let* n *be the length of the greatest common prefix of* d *and* d'*. We say* $\begin{smallmatrix}c_0\\d_0\end{smallmatrix}$ precedes $\begin{smallmatrix}c_1\\d_1\end{smallmatrix}$ *and write* $\begin{smallmatrix}c_0\\d_0\end{smallmatrix} \prec \begin{smallmatrix}c_1\\d_1\end{smallmatrix}$ *if the prefix of the length* n *of* c_0 *is lexically smaller than the prefix of length* n *of* c_1*. We say* $\begin{smallmatrix}c_0\\d_0\end{smallmatrix}$ *and* $\begin{smallmatrix}c_1\\d_1\end{smallmatrix}$ *are* aligned *if* $\begin{smallmatrix}c_0\\d_0\end{smallmatrix} \not\prec \begin{smallmatrix}c_1\\d_1\end{smallmatrix}$ *and* $\begin{smallmatrix}c_1\\d_1\end{smallmatrix} \not\prec \begin{smallmatrix}c_0\\d_0\end{smallmatrix}$.

Definition 4.4. *Let* c, c' *be control flow tags and* l, l' *be leakage of some type* L*. We say* (c, l) *and* (c', l') *satisfy* aligned indistinguishability *and write* $(c, l) \equiv (c', l')$ *if* $l = l'$ *and the control flow tags* c *and* c' *are aligned (see Definition 4.3). Aligned indistinguishability is a equivalence relation.*

Definition 4.5. *We say a program adheres to* aligned non-interference *with if it is non-interfering with respect to the equivalence relation of aligned indistinguishability.*

Assuming aligned non-interference we can now show that NIFTY guarantees horizontal consistency of local leakage transformers.

Lemma 4.4. *Let* p *be a source program satisfying aligned non-interference that is annotated by NIFTY. Then we have horizontal consistency of local leakage transformers for any two* public*-equal inputs.*

Proof Sketch. We prove this by instantiating a 2-simulation on the source traces using a synchroniser similar to the leakage synchroniser. In contrast to the leakage synchroniser, this synchroniser uses points of control flow divergence and rejoin as additional synchronised position. This guarantees both traces to be at the same program point when in *public* control flow. When in *secret* control flow, the sequence of local leakage transformers must be equal because any such branch is backtracked in NIFTY when they are not equal. □

Theorem 4.5. *For any program* p *that observes aligned non-interference and any optimisation and leakage model such that we have local leakage transformer simulations on* public*-equal inputs, the optimisation acting on the* p *annotated by NIFTY preserves non-interference.*

Proof. Apply Theorem 3.10 and use Theorem 4.4 for the horizontal consistency. □

$$\frac{Y \subseteq \{x\} \cup X \quad x \in Y \to \mathrm{live}_e\ e \subseteq X}{\mathrm{live}_{\mathrm{opt}}\,[x \leftarrow e] : (X, Y)}\ \text{opt.} \qquad \frac{Y \subseteq \{x\} \cup X \quad \mathrm{live}_e\ e \subseteq X \quad x \in Y}{\mathrm{live}_{\mathrm{fallback}}\,[x \leftarrow e] : (X, Y)}\ \text{fallback}$$

Fig. 9. Judgements showing how the optimising and the fallback liveness analyses act on an assignment.

4.3 Case Studies

We demonstrate the applicability of our approach by defining local leakage transformer simulations for three optimisations and then applying Theorem 4.2 and Theorem 4.5 on them to get proofs of CT-preservation and non-interference preservation for any given leakage model. This way, our approach allows to separate optimisation-specific arguments from security policy specific arguments which is not directly possible in related work. For each of the three optimisations we only have to show a local leakage transformer simulation and get CT-preservation and non-interference preservation for free through the theorems we have established in this paper. Note that while our proofs work for any leakage model, the aggressiveness of the optimisation that NIFTY performs will be different: a weaker leakage model allows for more aggressive optimisations.

Dead Code Elimination. Dead code elimination was used as a running example throughout the paper. To identify assignments to *dead* variables, i.e. variables not used before they are either overwritten or the program terminates, a *liveness* analysis is required. The liveness analysis annotates at every program point the *live-before set* X and the *live-out set* Y, denoting for variables whether they are live before and/or after the statement. We define two variants of it (see Fig. 9): First, the *optimising* variant incorporates possible eliminations by only requiring the variables of the expression to be live, if the assigned variable is live itself. This ensures that a dead assignment does not extend the lifetime of the variables used therein. In contrast, in the fallback approach the variables of the respective expression are required to be live either way and the assigned variable itself is considered to be live. Given a program with annotated liveness information, dead code elimination removes any assignment to a variable x such that x is not live at that program point. The local leakage transformers are defined analogously: We use `rm` whenever a leaky assignment is eliminated and `id` on every other leaky statement.

Lemma 4.6. *We have a local leakage transformer simulation for dead code elimination.*

Proof. We step through source and target program. Whenever an assignment is eliminated, we stall on the target trace and, if there is leakage, then the source is marked by `rm`, which is fulfilled. In every other case, source and target are processed in lockstep and any leakage is preserved. □

Corollary 4.7. *Dead code elimination preserves CT.*

Corollary 4.8. *Applying NIFTY on dead code elimination preserves non-interference.*

Common Branch Factorisation. Common branch factorisation (in short: CBF) does not need a data flow analysis like dead code elimination. Instead, it simply moves assignments at the start of both branches of a conditional outside of the conditional, if there is no conflicting dependence to the branch predicate. We still need an analysis for the correct annotation of local leakage transformers. As the optimisation may swap leakages—the one of the branch itself and the one of the moved assignment—, we need a little trick to be able to formulate local leakage transformers: We define an equivalent leakage model ℓ' that *merges* these two leakages A and B to one leakage event AB. Now, the data flow analysis flags any position that is admissible for a factorisation. The local leakages transformer at optimised program points reverses any leakage AB to BA. The fallback pass never flags anything as admissible, and the actual optimisation only optimises program points with a flag.

Lemma 4.9. *We have a local leakage transformer simulation for CBF.*

Proof. A simulation proof where at every optimised conditional the target trace processes the factorised assignment with the branch on source and the branch on the target trace is processed together with the assignment on source. □

Corollary 4.10. *Common branch factorisation preserves CT.*

Corollary 4.11. *Applying NIFTY on CBF preserves non-interference.*

Linearisation. Linearisation is a technique that replaces conditional control flow by a linear sequence of predicated instructions to save the overhead of branching. This pass is only interesting when branches *do* leak (otherwise no leakage is changed, non-interference preservation holds trivially), so we will only consider it for CT. Our linearisation pass is similar to the one discussed in [Barthe et al. 2020] in which the authors had to resort to a complex CT-simulation proof to support CompCert's branch placement heuristic in the linearised code. Our approach handles it without any additional effort. Here, we assume a data flow transformer that marks any conditional with *true* or *false*, signifying whether the consequence and alternative of the conditional should come first when linearised. The corresponding local leakage transformer maps booleans to booleans: It maps whether the branch goes left or right to whether there is jump or not. The actual optimisation then linearises the code according to the ordering annotation.

Lemma 4.12. *We have a local leakage transformer simulation for linearisation.*

Proof. This is a simple lockstep simulation where we show that the target trace jumps exactly when it is supposed to according to the local leakage transformer. □

Corollary 4.13. *Linearisation preserves CT.*

5 Related Work

Constant-time simulations (CT-simulations). Our work directly extends the work of Barthe et al. [2018], which presents a similar technique for proving the preservation of the *constant-time policy* through compiler passes. Our definition of hyperproperty simulation (Definition 3.8) generalises manysteps CT-simulation of Definition 8 of their work, and our formalisation of leakage is also based on their Section IV. However, our proof technique nontrivially extends their work by supporting proofs of preservation of k-hypersafety properties, not just the 2-trace constant-time policy. Technically, we generalise the admissible source-trace relations, which are denoted $\approx_{\text{src}}$ in our work and $\equiv_S$ in their work. In their work, such a relation must hold after every pair of steps in the two source traces (i.e., the two source traces are considered in lock-step). We relax this requirement by admitting relations between k source traces and allowing the prover to choose the positions at which $\equiv_{\text{src}}$ holds as long as no trace stalls forever (*cf.* Definition 3.3). This relaxation is necessary for representing properties like leakage-based noninterference which does not require the relation (indistinguishability) to hold at each step inside secret branches. The constant-time policy considered by Barthe et al. [2018] prohibits branching on secrets by definition, so they do not need this relaxation and do not consider it. Phrased differently in the terminology of Section III-C of Barthe et al. [2018], we support hyperproperties based on both "locally preserves" and "step-consistent" unwinding lemmas, whereas they only support properties based on former.

CT-simulations have been used to prove that the Jasmin compiler [Almeida et al. 2017], which was designed for low-level constant-time programming, preserves the constant time policy under various leakage models [Shivakumar et al. 2022]. CT-simulations, in conjunction with other techniques, have also been used to prove that the CompCert compiler [Blazy et al. 2006; Leroy 2006, 2009] can preserve

the constant-time policy with minimal modifications [Barthe et al. 2020]. While not validated on realistic compilers like Jasmin, our technique supports a broader class of hyperproperties. This idea has been extended for the preservation of speculative constant-time: [Olmos et al. 2025; van der Wall and Meyer 2025] use a cube diagram similar to CT-simulations and to our proposed hyperproperty simulations for preseving speculative constant-time. Again, in contrast to our work the simulation is lockstep on one of the trace pairs and does not support secret branches.

To ease proofs of the preservation of constant-time, [Barthe et al. 2021] introduce *structured leakage*: Here, the problem of the expressivity of a *global* leakage transformer is circumvented by making the leakage more expressive. Instead of interpreting leakage as a series of leakage events, the leakage of a program itself is of a tree structure, mimicing the program. In this setup, a global leakage transformer is sufficient, even for optimisations like dead code elimination. We achieve similar results for CT-preservation on the more common interpretation of leakage as a series of events. It would be interesting to investigate whether our annotations of local leakage transformers could be translated into a global leakage transformer for structured leakage and vice versa, and whether the idea of structured leakage could be incorporated into NIFTY.

Orthogonal to work on *preserving* constant-time through compilation is work on *enforcing* that a given program (in a single language, either source or target) respects constant-time. Constant-time is a simplification of noninterference, so it can be enforced by modifying a standard information flow type system for noninterference like Volpano et al. [1996]. A recent, industry-scale example of this approach is the design of CT-Wasm, an extension of Webassembly with a type system that enforces constant-time [Watt et al. 2019]. It would be interesting to apply techniques such as ours or CT-simulations to (simple) compilers that use CT-Wasm as the source or target language.

Preservation of noninterference and indistinguishability. Preservation of noninterference through compilation was first examined by [Barthe et al. 2007], who consider a source and target language, both equipped with noninterference-enforcing type systems. They prove that a (simple) compiler from source to target preserves well-typedness and, hence, well-typed source programs compile to noninterferent target programs. Although seminal, this approach rejects source programs that may be noninterferent but do not type check in the source and it does not consider optimisations. The approach also does not consider leakage or side-channels, concepts that became prominent years after the work was published. Consequently, the proof technique we develop (and the technique of Barthe et al. [2018]) are substantially different from this type-based technique, both in the mechanism and in the amenable sets of source and target languages.

The preservation and reflection of program indistinguishability (i.e., observation equivalence) through program transformations is also called full abstraction or, in some contexts, secure compilation [Patrignani et al. 2019]. This is a well-studied topic. Methods for establishing full abstraction include (bi)simulations on the operational semantics for untyped languages [Abadi 1998; Abadi et al. 2002], the use of sound and complete trace semantics to simplify the proofs [El-Korashy et al. 2021], and logical relations when the source and the target languages are typed [Ahmed and Blume 2008; Bowman and Ahmed 2015; Devriese et al. 2016]. However, these lines of work do not provide a systematic way of handling leakage. Our technique handles leakage and supports k-hyperproperties beyond indistinguishability.

Secure compilation. Recent work has proposed the preservation of classes of robust hyperproperties as a criterion for compiler security, where robust means "in all contexts". Abate et al. [2019] show that requiring a compiler to preserve increasingly larger classes of hyperproperties robustly results in increasingly stronger criteria for compilers. However, work on proof techniques to establish the preservation of robust hyperproperties has, so far, been limited to robust (1-trace) safety properties [Abate et al. 2018; El-Korashy et al. 2022; Patrignani and Garg 2021] and observational

equivalence/noninterference, which we explained above. Extending our approach to handle all contexts (robustness) would be an avenue for future work.

6 Conclusion & Future Work

In this work, we introduced a novel framework for hyperproperty preservation proofs, leveraging the concept of *k-simulations* to model simulations across an arbitrary number of traces and enabling the composition of complex simulations from simpler ones. Building on this, we developed *hyperproperty simulations* to facilitate modular proofs of hyperproperty preservation, particularly for indistinguishability properties like non-interference. We proposed the notion of *indistinguishability simulations*, which combine source-target leakage simulations with indistinguishability on source traces, simplifying cryptographic constant-time preservation proofs and enabling proofs for more general notions of non-interference. To address the challenge of non-interference preservation in compiler optimisations, we presented the *NIFTY* framework, which ensures non-interference preservation by selectively undoing optimisations in regions where they violate security guarantees. Our approach is proven correct for any optimisation and leakage model, given a source-target simulation proof for the specific case.

Our contributions invite for different directions in future work: The most obvious possible direction would be to extend our case study of NIFTY with different compilation passes or apply it to more realistic languages and semantics. Preservation of different security properties, for example using an explicit timing model, could be another research target. This is a challenge, as here the leakage has a *state*—the time passed since the last leakage event. As NIFTY already does a data flow analysis, it should be possible to extend the framework to support this as well. As the hyperproperty simulations are not specific to security policies, they could spark research in different domains that are interested in preservation of hyperproperties during compilation. For example, they may be employed to verify properties on single-program multiple-data (SPMD) programs.

Data-Availability Statement

This work has an accompanying Rocq development which has been submitted as an artifact [Rosemann et al. 2025]. All theorems, lemmas and corollaries in this paper are formalised. Due to the nature of formalised proofs, many statements are described in a much more abstract way in the paper. The development consists of roughly 15k lines of code. Most of it is for supporting the definitions and propositions in Sections 3, 4.1 and 4.2. The case studies of Section 4.3 require significantly less code.

State of the Proofs in the Submitted Artifact. All statements in Section 3 are proven completely (except for very minor technical lemmas). The NIFTY framework is completely defined in Rocq and its correctness properties are stated but some of their proofs are incomplete. For the case studies in Section 4.3 the relevant data flow transformers (used by NIFTY) are defined but the proofs are incomplete.

References

Martín Abadi. 1998. Protection in Programming-Language Translations. In *Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 13-17, 1998, Proceedings (Lecture Notes in Computer Science, Vol. 1443)*, Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel (Eds.). Springer, 868–883. doi:10.1007/BFB0055109

Martín Abadi, Cédric Fournet, and Georges Gonthier. 2002. Secure Implementation of Channel Abstractions. *Inf. Comput.* 174, 1 (2002), 37–83. doi:10.1006/INCO.2002.3086

Carmine Abate, Roberto Blanco, Deepak Garg, Catalin Hritcu, Marco Patrignani, and Jeremy Thibault. 2019. Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. IEEE. doi:10.1109/csf.2019.00025

Carmine Abate, Arthur Azevedo de Amorim, Roberto Blanco, Ana Nora Evans, Guglielmo Fachini, Catalin Hritcu, Théo Laurent, Benjamin C. Pierce, Marco Stronati, and Andrew Tolmach. 2018. When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 1351–1368. doi:10.1145/3243734.3243745

Amal Ahmed and Matthias Blume. 2008. Typed closure conversion preserves observational equivalence. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, James Hook and Peter Thiemann (Eds.). ACM, 157–168. doi:10.1145/1411204.1411227

José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. 2017. Jasmin: High-Assurance and High-Speed Cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 1807–1823. doi:10.1145/3133956.3134078

Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. 2020. Formal verification of a constant-time preserving C compiler. *Proc. ACM Program. Lang.* 4, POPL (2020), 7:1–7:30. doi:10.1145/3371075

Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. 2018. Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time". In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*. IEEE Computer Society, 328–343. doi:10.1109/CSF.2018.00031

Gilles Barthe, Benjamin Grégoire, Vincent Laporte, and Swarn Priya. 2021. Structured Leakage and Applications to Cryptographic Constant-Time and Cost. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 462–476. doi:10.1145/3460120.3484761

Gilles Barthe, Tamara Rezk, and Amitabh Basu. 2007. Security types preserving compilation. *Comput. Lang. Syst. Struct.* 33, 2 (2007), 35–59. doi:10.1016/J.CL.2005.05.002

Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. 2006. Formal Verification of a C Compiler Front-End. In *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4085)*, Jayadev Misra, Tobias Nipkow, and Emil Sekerinski (Eds.). Springer, 460–475. doi:10.1007/11813040_31

Aaron Bohannon, Benjamin C. Pierce, Vilhelm Sjöberg, Stephanie Weirich, and Steve Zdancewic. 2009. Reactive noninterference. In *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*, Ehab Al-Shaer, Somesh Jha, and Angelos D. Keromytis (Eds.). ACM, 79–90. doi:10.1145/1653662.1653673

William J. Bowman and Amal Ahmed. 2015. Noninterference for free. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, Kathleen Fisher and John H. Reppy (Eds.). ACM, 101–113. doi:10.1145/2784731.2784733

Ethan Cecchetti, Andrew C. Myers, and Owen Arden. 2017. Nonmalleable Information Flow Control. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 1875–1891. doi:10.1145/3133956.3134054

Kevin Cheang, Cameron Rasmussen, Sanjit A. Seshia, and Pramod Subramanyan. 2019. A Formal Approach to Secure Speculation. In *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*. IEEE, 288–303. doi:10.1109/CSF.2019.00027

Dominique Devriese, Marco Patrignani, and Frank Piessens. 2016. Fully-abstract compilation by approximate back-translation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 164–177. doi:10.1145/2837614.2837618

Akram El-Korashy, Roberto Blanco, Jérémy Thibault, Adrien Durier, Deepak Garg, and Catalin Hritcu. 2022. SecurePtrs: Proving Secure Compilation with Data-Flow Back-Translation and Turn-Taking Simulation. In *35th IEEE Computer Security Foundations Symposium, CSF 2022, Haifa, Israel, August 7-10, 2022*. IEEE, 64–79. doi:10.1109/CSF54842.2022.9919680

Akram El-Korashy, Stelios Tsampas, Marco Patrignani, Dominique Devriese, Deepak Garg, and Frank Piessens. 2021. CapablePtrs: Securely Compiling Partial Programs Using the Pointers-as-Capabilities Principle. In *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021*. IEEE, 1–16. doi:10.1109/CSF51468.2021.00036

Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, J. Gregory Morrisett and Simon L. Peyton Jones (Eds.). ACM, 42–54.

doi:10.1145/1111037.1111042

Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. doi:10.1145/1538788.1538814

Kedar S. Namjoshi. 1997. A Simple Characterization of Stuttering Bisimulation. In *Foundations of Software Technology and Theoretical Computer Science, 17th Conference, Kharagpur, India, December 18-20, 1997, Proceedings (Lecture Notes in Computer Science, Vol. 1346)*, S. Ramesh and G. Sivakumar (Eds.). Springer, 284–296. doi:10.1007/BFB0058037

Santiago Arranz Olmos, Gilles Barthe, Lionel Blatter, Benjamin Grégoire, and Vincent Laporte. 2025. Preservation of Speculative Constant-Time by Compilation. *Proc. ACM Program. Lang.* 9, POPL (2025), 1293–1325. doi:10.1145/3704880

Marco Patrignani, Amal Ahmed, and Dave Clarke. 2019. Formal Approaches to Secure Compilation: A Survey of Fully Abstract Compilation and Related Work. *ACM Comput. Surv.* 51, 6 (2019), 125:1–125:36. doi:10.1145/3280984

Marco Patrignani and Deepak Garg. 2021. Robustly Safe Compilation, an Efficient Form of Secure Compilation. *ACM Trans. Program. Lang. Syst.* 43, 1 (2021), 1:1–1:41. doi:10.1145/3436809

Julian Rosemann, Sebastian Hack, and Deepak Garg. 2025. *Non-interference Preserving Optimising Compilation.* doi:10.5281/zenodo.16929228

Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, and Swarn Priya. 2022. Enforcing Fine-grained Constant-time Policies. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.). ACM, 83–96. doi:10.1145/3548606.3560689

Sören van der Wall and Roland Meyer. 2025. SNIP: Speculative Execution and Non-Interference Preservation for Compiler Transformations. *Proc. ACM Program. Lang.* 9, POPL (2025), 1506–1535. doi:10.1145/3704887

Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. 1996. A Sound Type System for Secure Flow Analysis. *J. Comput. Secur.* 4, 2/3 (1996), 167–188. doi:10.3233/JCS-1996-42-304

Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. 2019. CT-wasm: type-driven secure cryptography for the web ecosystem. *Proc. ACM Program. Lang.* 3, POPL (2019), 77:1–77:29. doi:10.1145/3290390

Hans Winderix, Marton Bognar, Lesly-Ann Daniel, and Frank Piessens. 2024. Libra: Architectural Support For Principled, Secure And Efficient Balanced Execution On High-End Processors. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*, Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie (Eds.). ACM, 19–33. doi:10.1145/3658644.3690319

Received 2025-03-25; accepted 2025-08-12