

Trace-Relating Compiler Correctness and Secure Compilation

Carmine Abate¹ Roberto Blanco¹ Ștefan Ciobâcă² Adrien Durier¹
Deepak Garg³ Cătălin Hrițcu¹ Marco Patrignani^{4,5} Éric Tanter^{6,1} Jérémy Thibault¹

¹Inria Paris, France ²UAIC Iași, Romania ³MPI-SWS, Saarbrücken, Germany ⁴Stanford University, Stanford, USA
⁵CISPA, Saarbrücken, Germany ⁶University of Chile, Santiago, Chile

Abstract. Compiler correctness is, in its simplest form, defined as the inclusion of the set of traces of the compiled program into the set of traces of the original program, which is equivalent to the preservation of all trace properties. Here traces collect, for instance, the externally observable events of each execution. This definition requires, however, the set of traces of the source and target languages to be exactly the same, which is not the case when the languages are far apart or when observations are fine-grained. To overcome this issue, we study a generalized compiler correctness definition, which uses source and target traces drawn from potentially different sets and connected by an arbitrary relation. We set out to understand what guarantees this generalized compiler correctness definition gives us when instantiated with a non-trivial relation on traces. When this trace relation is not equality, it is no longer possible to preserve the trace properties of the source program unchanged. Instead, we provide a generic characterization of the target trace property ensured by correctly compiling a program that satisfies a given source property, and dually, of the source trace property one is required to show in order to obtain a certain target property for the compiled code. We show that this view on compiler correctness can naturally account for undefined behavior, resource exhaustion, different source and target values, side-channels, and various abstraction mismatches. Finally, we show that the same generalization also applies to many secure compilation definitions, which characterize the protection of a compiled program against linked adversarial code.

1 Introduction

Compiler correctness is an old idea [38, 41, 42] that has seen a significant revival in recent times. This new wave was started by the creation of the CompCert verified C compiler [34] and continued by the proposal of many significant extensions and variants of CompCert [8, 9, 12, 23, 30, 31, 43, 56, 61, 63, 67] and the success of many other milestone compiler verification projects, including Vellvm [70], Pilsner [46], CakeML [64], CertiCoq [4], etc. Yet, even for these verified compilers, the precise statement of correctness matters. Since proof assistants are used to conduct the verification, an external observer does not have to understand the proofs in order to trust them, but one still has to deeply understand the statement that was proved. And this is true not just for correct compilation, but also for secure compilation, which is the more recent idea that our compilation chains should do more to also ensure security of our programs [3, 26].

Basic Compiler Correctness. The gold standard for compiler correctness is *semantic preservation*, which intuitively says that the semantics of a compiled program (in the target language) is compatible with the semantics of the original program (in the source

language). For practical verified compilers, such as CompCert [34] and CakeML [64], semantic preservation is stated extrinsically, by referring to *traces*. In these two settings, a trace is an ordered sequence of events—such as inputs from and outputs to an external environment—that are produced by the execution of a program.

A basic definition of compiler correctness can be given by the set inclusion of the traces of the compiled program into the traces of the original program. Formally [34]:

Definition 1.1 (Basic Compiler Correctness (CC)). *A compiler \downarrow is correct iff*

$$\forall W t. W \downarrow \rightsquigarrow t \Rightarrow W \rightsquigarrow t.$$

This definition says that for any whole¹ source program W , if we compile it (denoted $W \downarrow$), execute it with respect to the semantics of the target language, and observe a trace t , then the original W can produce *the same* trace t with respect to the semantics of the source language.² This definition is simple and easy to understand, since it only references a few familiar concepts: a compiler between a source and a target language, each equipped with a trace-producing semantics (usually nondeterministic).

Beyond Basic Compiler Correctness. This basic compiler correctness definition assumes that any trace produced by a compiled program can be produced by the source program. This is a very strict requirement, and in particular implies that the source and target traces are drawn from the same set and that the same source trace corresponds to a given target trace. These assumptions are often too strong, and hence in practice verified compiler efforts use different formulations of compiler correctness:

CompCert [34] The original compiler correctness theorem of CompCert [34] can be seen as an instance of basic compiler correctness, but it does not provide any guarantees for programs that can exhibit undefined behavior [57]. As allowed by the C standard, such unsafe programs are not even considered to be in the source language, so are not quantified over. This has important practical implications, since undefined behavior often leads to exploitable security vulnerabilities [13, 24, 25] and serious confusion even among experienced C and C++ developers [33, 57, 65, 66]. As such, since 2010, CompCert provides an additional top-level correctness theorem³ that better accounts for the presence of unsafe programs by providing guarantees for them up to the point when they encounter undefined behavior [57]. This new theorem goes beyond the basic correctness definition above, as a target trace need only correspond to a source trace *up to the occurrence* of undefined behavior in the source trace.

CakeML [64] Compiler correctness for CakeML accounts for memory exhaustion in target executions. Crucially, memory exhaustion events cannot occur in source traces, only in target traces. Hence, dually to CompCert, compiler correctness only requires source and target traces to coincide up to the occurrence of a memory exhaustion event in the target trace.

¹ For simplicity, for now we ignore separate compilation and linking, returning to it in §5.

² Typesetting convention [50]: we use a **blue, sans-serif** font for **source** elements, an **orange, bold** font for **target** ones and a *black, italic* font for elements common to both languages.

³ Stated at the top of the CompCert file `driver/Complements.v` and discussed by Regehr [57].

Trace-Relating Compiler Correctness. Generalized formalizations of compiler correctness like the ones above can be naturally expressed as instances of a uniform definition, which we call *trace-relating compiler correctness*. This generalizes basic compiler correctness by (a) considering that source and target traces belong to *possibly distinct* sets \mathbf{Traces}_S and \mathbf{Trace}_T , and (b) being parameterized by an arbitrary *trace relation* \sim .

Definition 1.2 (Trace-Relating Compiler Correctness (CC^\sim)). A compiler \downarrow is correct with respect to a trace relation $\sim \subseteq \mathbf{Traces}_S \times \mathbf{Trace}_T$ iff

$$\forall W. \forall t. W \downarrow \rightsquigarrow t \Rightarrow \exists s \sim t. W \rightsquigarrow s.$$

This definition requires that, for any target trace t produced by the compiled program $W \downarrow$, there exist a source trace s that can be produced by the original program W and is *related* to t according to \sim (i.e., $s \sim t$). By choosing the trace relation appropriately, one can recover the different notions of compiler correctness presented above:

Basic CC Take $s \sim t$ to be $s = t$. Trivially, the basic CC of Definition 1.1 is $CC^=$.

CompCert Undefined behavior is modeled in CompCert as a trace-terminating event *Goes_wrong* that can occur in any of its languages (source, target, and all intermediate languages), so for a given phase (or composition thereof), we have $\mathbf{Traces}_S = \mathbf{Trace}_T$. Nevertheless, the relation between source and target traces with which to instantiate CC^\sim to obtain CompCert’s current theorem is:

$$s \sim t \equiv s = t \vee (\exists m \leq t. s = m \cdot \mathit{Goes_wrong}).$$

A compiler satisfying CC^\sim for this trace relation can turn a source trace ending in undefined behavior $m \cdot \mathit{Goes_wrong}$ (where “ \cdot ” is concatenation) either into the same trace in the target (first disjunct), or into a target trace that starts with the prefix m but then continues *arbitrarily* (second disjunct, “ \leq ” is the prefix relation).

CakeML Here, target traces are sequences of symbols from an alphabet Σ_T that has a specific trace-terminating event, **Resource_limit_hit**, which is not available in the source alphabet Σ_S (i.e., $\Sigma_T = \Sigma_S \cup \{\mathbf{Resource_limit_hit}\}$). Then, the compiler correctness theorem of CakeML can be obtained by instantiating CC^\sim with the following \sim relation:

$$s \sim t \equiv s = t \vee (\exists m. m \leq s. t = m \cdot \mathbf{Resource_limit_hit}).$$

The resulting CC^\sim instance relates a target trace ending in **Resource_limit_hit** after executing m to a source trace that first produces m and then continues in a way given by the semantics of the source program.

Beyond undefined behavior and resource exhaustion, there are many other practical uses for CC^\sim : in this paper we show that it also accounts for differences between source and target values, for a single source output being turned into a series of target outputs, and for side-channels.

On the flip side, the compiler correctness statement and its implications can be more difficult to understand for CC^\sim than for $CC^=$. The full implications of choosing a particular \sim relation can be subtle. In fact, using a bad relation can make the compiler correctness statement trivial or unexpected. For instance, it should be easy to see that if one uses the total relation, which relates all source traces to all target ones, the CC^\sim property holds for every compiler, yet it might take one a bit more effort to understand that the same is true even for the following relation:

$$s \sim t \equiv \exists W. W \rightsquigarrow s \wedge W \downarrow \rightsquigarrow t.$$

Reasoning About Trace Properties. To understand more about a particular CC^\sim instance, we propose to also look at how it preserves *trace properties*—defined as sets of allowed traces [32]—from the source to the target. For instance, it is well known that $CC^=$ is equivalent to the preservation of all trace properties (where $W \models \pi$ reads “ W satisfies π ” and stands for $\forall t. W \rightsquigarrow t \Rightarrow t \in \pi$):

$$CC^= \equiv \forall \pi \in 2^{\text{Trace}} \forall W. W \models \pi \Rightarrow W \downarrow \models \pi.$$

However, to the best of our knowledge, similar results have not been formulated for trace relations beyond equality, when it is no longer possible to preserve the trace properties of the source program unchanged. For trace-relating compiler correctness, where source and target traces can be drawn from different sets and related by an arbitrary trace relation, there are two crucial questions to ask:

1. For a source trace property π_S of a program—established for instance by formal verification—what is the strongest target property that any CC^\sim compiler is guaranteed to ensure for the produced target program?
2. For a target trace property π_T , what is the weakest source property we need to show of the original source program to obtain π_T for the result of any CC^\sim compiler?

Far from being mere hypothetical questions, they can help the developer of a verified compiler to better understand the compiler correctness theorem they are proving, and we expect that any user of such a compiler will need to ask either one or the other if they are to make use of that theorem. In this work we provide a simple and natural answer to these questions, for any instance of CC^\sim . Building upon a bijection between relations and Galois connections [5, 20, 44], we observe that any trace relation \sim corresponds to two *property mappings* $\tilde{\tau}$ and $\tilde{\sigma}$, which are functions mapping source properties to target ones ($\tilde{\tau}$ standing for “to target”) and target properties to source ones ($\tilde{\sigma}$ standing for “to source”):

$$\tilde{\tau}(\pi_S) = \{t \mid \exists s. s \sim t \wedge s \in \pi_S\}; \quad \tilde{\sigma}(\pi_T) = \{s \mid \forall t. s \sim t \Rightarrow t \in \pi_T\}.$$

The *existential image* of \sim , $\tilde{\tau}$, answers the first question above by mapping a given source property π_S to the target property that contains all target traces for which *there exists a related source trace* that satisfies π_S . Dually, the *universal image* of \sim , $\tilde{\sigma}$, answers the second question by mapping a given target property π_T to the source property that contains all source traces for which *all related target traces* satisfy π_T . We introduce two new correct compilation definitions in terms of *trace property preservation* (TP): $TP^{\tilde{\tau}}$ quantifies over all source trace properties and uses $\tilde{\tau}$ to obtain the corresponding target properties. $TP^{\tilde{\sigma}}$ quantifies over all target trace properties and uses $\tilde{\sigma}$ to obtain the corresponding source properties. We prove that these two definitions are equivalent to CC^\sim , yielding a novel trinitarian view of compiler correctness (Figure 1).


$$\begin{array}{c}
 \forall W. \forall t. W \downarrow \rightsquigarrow t \Rightarrow \exists s \sim t. W \rightsquigarrow s \\
 \parallel \\
 CC^\sim \\
 \begin{array}{ccc}
 \forall \pi_T. \forall W. W \models \tilde{\sigma}(\pi_T) & & \forall \pi_S. \forall W. W \models \pi_S \\
 \Rightarrow W \downarrow \models \pi_T & \equiv & TP^{\tilde{\sigma}} \xleftrightarrow{\quad} TP^{\tilde{\tau}} \equiv \Rightarrow W \downarrow \models \tilde{\tau}(\pi_S)
 \end{array}
 \end{array}$$

Fig. 1: The equivalent compiler correctness definitions forming our trinitarian view.

Contributions.

- ▶ We propose a new trinitarian view of compiler correctness that accounts for non-trivial trace relations. While, as discussed above, specific instances of the CC^\sim definition have already been used in practice, we seem to be the first to propose assessing the meaningfulness of CC^\sim instances in terms of how properties are preserved between the source and the target, and in particular by looking at the property mappings $\tilde{\sigma}$ and $\tilde{\tau}$ induced by the trace relation \sim . We prove that CC^\sim , $TP^{\tilde{\sigma}}$, and $TP^{\tilde{\tau}}$ are equivalent for any trace relation (§2.2), as illustrated in Figure 1. In the opposite direction, we show that for every trace relation corresponding to a given Galois connection [20], an analogous equivalence holds. Finally, we extend these results (§2.3) from the preservation of trace properties to the larger class of subset-closed hyperproperties (e.g., noninterference).
- ▶ We use CC^\sim compilers of various complexities to illustrate that our view on compiler correctness naturally accounts for undefined behavior (§3.1), resource exhaustion (§3.2), different source and target values (§3.3), and differences in the granularity of data and observable events (§3.4). We expect these ideas to apply to any other discrepancies between source and target traces. For each compiler we show how to choose the relation between source and target traces and how the induced property mappings preserve interesting trace properties and subset-closed hyperproperties. We look at the way particular $\tilde{\sigma}$ and $\tilde{\tau}$ work on different kinds of properties and how the produced properties can be expressed for different kinds of traces.
- ▶ We analyze the impact of correct compilation on noninterference [22], showing what can still be preserved (and thus also what is lost) when target observations are finer than source ones, e.g., side-channel observations (§4). We formalize the guarantee obtained by correct compilation of a noninterfering program as *abstract noninterference* [21], a weakening of target noninterference. Dually, we identify a family of declassifications of target noninterference for which source reasoning is possible.
- ▶ Finally, we show that the trinitarian view also extends to a large class of *secure compilation* definitions [2], formally characterizing the protection of the compiled program against linked adversarial code (§5). For each secure compilation definition we again propose both a property-free characterization in the style of CC^\sim , and two characterizations in terms of preserving a class of source or target properties satisfied against arbitrary adversarial contexts. The additional quantification over contexts allows for finer distinctions when considering different property classes, so we study mapping classes not only of trace properties and hyperproperties, but also of relational hyperproperties [2]. An example secure compiler accounting for a target that can produce additional trace events that are not possible in the source illustrates this approach.

The paper closes with discussions of related (§6) and future work (§7). The appendix contains omitted technical details.

The traces considered in our examples are structured, usually as sequences of events. We notice however that unless explicitly mentioned, all our definitions and results are more general and make no assumption whatsoever about the structure of traces. Most of the theorems formally or informally mentioned in the paper were mechanized in the Coq proof assistant and are marked with . This development has around 10k lines of code, is described in the online appendix, and is available at the following address:

https://github.com/secure-compilation/different_traces.

2 Trace-Relating Compiler Correctness

In this section, we start by generalizing the trace property preservation definitions at the end of the introduction to TP^σ and TP^τ , which depend on two *arbitrary* mappings σ and τ (§2.1). We prove that, whenever σ and τ form a Galois connection, TP^σ and TP^τ are equivalent (Theorem 2.4). We then exploit a bijective correspondence between trace relations and Galois connections to close the trinitarian view (§2.2), with two main benefits: first, it helps us assess the meaningfulness of a given trace relation by looking at the property mappings it induces; second, it allows us to construct new compiler correctness definitions starting from a desired mapping of properties. Finally, we generalize the classic result that compiler correctness (i.e., CC^-) is enough to preserve not just trace properties but also all subset-closed hyperproperties [14]. For this, we show that CC^\sim is also equivalent to subset-closed hyperproperty preservation, for which we also define both a version in terms of $\tilde{\sigma}$ and a version in terms of $\tilde{\tau}$ (§2.3).

2.1 Property Mappings

As explained in §1, trace-relating compiler correctness CC^\sim , by itself, lacks a crisp description of which trace properties are preserved by compilation. Since even the syntax of traces can differ between source and target, one can either look at trace properties of the source (but then one needs to interpret them in the target), or at trace properties of the target (but then one needs to interpret them in the source). Formally we need two property mappings, $\tau : 2^{\text{Traces}_S} \rightarrow 2^{\text{Trace}_T}$ and $\sigma : 2^{\text{Trace}_T} \rightarrow 2^{\text{Traces}_S}$, which lead us to the following generalization of trace property preservation (TP).

Definition 2.1 (TP^σ and TP^τ). *Given two property mappings, $\tau : 2^{\text{Traces}_S} \rightarrow 2^{\text{Trace}_T}$ and $\sigma : 2^{\text{Trace}_T} \rightarrow 2^{\text{Traces}_S}$, for a compilation chain $\cdot\downarrow$ we define:*

$$\text{TP}^\tau \equiv \forall \pi_S. \forall W. W \models \pi_S \Rightarrow W\downarrow \models \tau(\pi_S); \quad \text{TP}^\sigma \equiv \forall \pi_T. \forall W. W \models \sigma(\pi_T) \Rightarrow W\downarrow \models \pi_T.$$

For an arbitrary source program W , τ interprets a source property π_S as the *target guarantee* for $W\downarrow$. Dually, σ defines a *source obligation* sufficient for the satisfaction of a target property π_T after compilation. Ideally:

- Given π_T , the target interpretation of the source obligation $\sigma(\pi_T)$ should actually guarantee that π_T holds, i.e., $\tau(\sigma(\pi_T)) \subseteq \pi_T$;
- Dually for π_S , we would not want the source obligation for $\tau(\pi_S)$ to be harder than π_S itself, i.e., $\sigma(\tau(\pi_S)) \supseteq \pi_S$.

These requirements are satisfied when the two maps form a *Galois connection* between the posets of source and target properties ordered by inclusion. We briefly recall the definition and the characteristic property of Galois connections [16, 39].

Definition 2.2 (Galois connection). *Let (X, \preceq) and (Y, \sqsubseteq) be two posets. A pair of maps, $\alpha : X \rightarrow Y, \gamma : Y \rightarrow X$ is a Galois connection iff it satisfies the adjunction law: $\forall x \in X. \forall y \in Y. \alpha(x) \sqsubseteq y \iff x \preceq \gamma(y)$. α (resp. γ) is the lower (upper) adjoint or abstraction (concretization) function and Y (X) the abstract (concrete) domain.*

We will often write $\alpha : (X, \preceq) \sqsupseteq (Y, \sqsubseteq) : \gamma$ to denote a Galois connection, or simply $\alpha : X \sqsupseteq Y : \gamma$, or even $\alpha \sqsupseteq \gamma$ when the involved posets are clear from context.

Lemma 2.3 (Characteristic property of Galois connections). *If $\alpha:(X, \preceq) \sqsupseteq (Y, \sqsubseteq) : \gamma$ is a Galois connection, then α, γ are monotone and they satisfy these properties:*

$$i) \quad \forall x \in X. x \preceq \gamma(\alpha(x)); \quad ii) \quad \forall y \in Y. \alpha(\gamma(y)) \sqsubseteq y.$$

If X, Y are complete lattices, then α is continuous, i.e., $\forall F \subseteq X. \alpha(\bigsqcup F) = \bigsqcup \alpha(F)$.

If two property mappings, τ and σ , form a Galois connection on trace properties ordered by set inclusion, Lemma 2.3 (with $\alpha = \tau$ and $\gamma = \sigma$) tells us that they satisfy the ideal conditions we discussed above, i.e., $\tau(\sigma(\pi_{\mathbf{T}})) \subseteq \pi_{\mathbf{T}}$ and $\sigma(\tau(\pi_{\mathbf{S}})) \supseteq \pi_{\mathbf{S}}$.⁴

The two ideal conditions on τ and σ are sufficient to show the equivalence of the criteria they define, respectively TP^τ and TP^σ .

Theorem 2.4 (TP $^\tau$ and TP $^\sigma$ coincide \Leftrightarrow). *Let $\tau : 2^{\text{Traces}} \rightleftharpoons 2^{\text{TraceT}} : \sigma$ be a Galois connection, with τ and σ the lower and upper adjoints (resp.). Then $\text{TP}^\tau \iff \text{TP}^\sigma$.*

2.2 Trace Relations and Property Mappings

We now investigate the relation between CC^\sim , TP^τ and TP^σ . We show that for a trace relation and its corresponding Galois connection (Lemma 2.7), the three criteria are equivalent (Theorem 2.8). This equivalence offers interesting insights for both verification and design of a correct compiler. For a CC^\sim compiler, the equivalence makes explicit both the guarantees one has after compilation ($\tilde{\tau}$) and source proof obligations to ensure the satisfaction of a given target property ($\tilde{\sigma}$). On the other hand, a compiler designer might first determine the target guarantees the compiler itself must provide, i.e., τ , and then prove an equivalent statement, CC^\sim , for which more convenient proof techniques exist in the literature [7, 64].

Definition 2.5 (Existential and Universal Image [20]). *Given any two sets X and Y and a relation $\sim \subseteq A \times B$, define its existential or direct image, $\tilde{\tau} : 2^X \rightarrow 2^Y$ and its universal image, $\tilde{\sigma} : 2^Y \rightarrow 2^X$ as follows:*

$$\tilde{\tau} = \lambda \pi \in 2^X. \{y \mid \exists x. x \sim y \wedge x \in \pi\}; \quad \tilde{\sigma} = \lambda \pi \in 2^Y. \{x \mid \forall y. x \sim y \Rightarrow y \in \pi\}.$$

When trace relations are considered, the existential and universal images can be used to instantiate Definition 2.1 leading to the trinitarian view already mentioned in §1.

Theorem 2.6 (Trinitarian View \Leftrightarrow). *For any trace relation \sim and its existential and universal images $\tilde{\tau}$ and $\tilde{\sigma}$, we have: $\text{TP}^{\tilde{\tau}} \iff \text{CC}^\sim \iff \text{TP}^{\tilde{\sigma}}$.*

This result relies both on Theorem 2.4 and on the fact that the existential and universal images of a trace relation form a Galois connection (\Leftrightarrow). Below we further generalize this result (Theorem 2.8) relying on a bijective correspondence between trace relations and Galois connections on properties.

Lemma 2.7 (Trace relations \cong Galois connections on trace properties). *The function $\sim \mapsto \tilde{\tau} \sqsupseteq \tilde{\sigma}$ that maps a trace relation to its existential and universal images is a bijection between trace relations $2^{\text{Traces}} \times \text{TraceT}$ and Galois connections on trace properties $2^{\text{Traces}} \sqsupseteq 2^{\text{TraceT}}$. Its inverse is $\tau \sqsupseteq \sigma \mapsto \tilde{\sim}$, where $s \tilde{\sim} t \equiv t \in \tau(\{s\})$.*

⁴ While target traces are often “more concrete” than source ones, trace properties 2^{Trace} (which in Coq we represent as the function type $\text{Trace} \rightarrow \text{Prop}$) are contravariant in Trace and thus target properties correspond to the *abstract domain*.

Proof. Gardiner et al. [20] show that the existential image is a functor from the category of sets and relations to the category of predicate transformers, mapping a set $X \mapsto 2^X$ and a relation $\sim \subseteq X \times Y \mapsto \tilde{\tau} : 2^X \rightarrow 2^Y$. They also show that such a functor is an isomorphism – hence bijective – when one considers only monotonic predicate transformers that have a – unique – upper adjoint. The universal image of \sim , $\tilde{\sigma}$, is the unique adjoint of $\tilde{\tau}$ ($\tilde{\sigma}$), hence $\sim \mapsto \tilde{\tau} \stackrel{\circlearrowleft}{\rightleftharpoons} \tilde{\sigma}$ is itself bijective. \square

The bijection just introduced allows us to generalize [Theorem 2.6](#) and switch between the three views of compiler correctness described earlier at will.

Theorem 2.8 (Correspondence of Criteria). *For any trace relation \sim and corresponding Galois connection $\tau \stackrel{\circlearrowleft}{\rightleftharpoons} \sigma$, we have: $\text{TP}^\tau \iff \text{CC}^\sim \iff \text{TP}^\sigma$.*

Proof. For a trace relation \sim and the Galois connection $\tilde{\tau} \stackrel{\circlearrowleft}{\rightleftharpoons} \tilde{\sigma}$, the result follows from [Theorem 2.6](#). For a Galois connection $\tau \stackrel{\circlearrowleft}{\rightleftharpoons} \sigma$ and $\hat{\sim}$, use [Lemma 2.7](#) to conclude that the existential and universal images of $\hat{\sim}$ coincide with τ and σ , respectively; the goal then follows from [Theorem 2.6](#). \square

We conclude by explicitly noting that sometimes the lifted properties may be trivial: the target guarantee can be the true property (the set of all traces), or the source obligation the false property (the empty set of traces). This might be the case when source observations abstract away too much information ([§3.2](#) presents an example).

2.3 Preservation of Subset-Closed Hyperproperties

A CC^\sim compiler ensures the preservation not only of trace properties, but also of all subset-closed hyperproperties, which are known to be preserved by refinement [14]. An example of a subset-closed hyperproperty is *noninterference* [14]; a CC^\sim compiler thus guarantees that if \mathbb{W} is noninterfering with respect to the inputs and outputs in the trace then so is $\mathbb{W}\downarrow$. To be able to talk about how (hyper)properties such as noninterference are preserved, in this section we propose another trinitarian view involving CC^\sim and preservation of subset-closed hyperproperties ([Theorem 2.11](#)), slightly weakened in that source and target property mappings will need to be closed under subsets.

First, recall that a program satisfies a hyperproperty when its complete set of traces, which from now on we will call its *behavior*, is a member of the hyperproperty [14].

Definition 2.9 (Hyperproperty Satisfaction). *A program W satisfies a hyperproperty H , written $W \models H$, iff $\text{beh}(W) \in H$, where $\text{beh}(W) = \{t \mid W \rightsquigarrow t\}$.*

Hyperproperty preservation is a strong requirement in general. Fortunately, many interesting hyperproperties are *subset-closed* (*SCH* for short), which simplifies their preservation since it suffices to show that the behaviors of the compiled program refine the behaviors of the source one, which coincides with the statement of CC^\sim .

To talk about hyperproperty preservation in the trace-relating setting, we need an interpretation of source hyperproperties into the target and vice versa. The one we consider builds on top of the two trace property mappings τ and σ , which are naturally lifted to hyperproperty mappings. This way we are able to extract two hyperproperty mappings from a trace relation similarly to [§2.2](#):

Definition 2.10 (Lifting property mappings to hyperproperty mappings). Let $\tau : 2^{\text{Traces}} \rightarrow 2^{\text{Trace}_T}$ and $\sigma : 2^{\text{Trace}_T} \rightarrow 2^{\text{Traces}}$ be arbitrary property mappings. The images of $\mathbf{H}_S \in 2^{2^{\text{Traces}}}$, $\mathbf{H}_T \in 2^{2^{\text{Trace}_T}}$ under τ and σ are, respectively:

$$\tau(\mathbf{H}_S) = \{\tau(\pi_S) \mid \pi_S \in \mathbf{H}_S\}; \quad \sigma(\mathbf{H}_T) = \{\sigma(\pi_T) \mid \pi_T \in \mathbf{H}_T\}.$$

Formally we are defining two new mappings, this time on hyperproperties, but by a small abuse of notation we still denote them by τ and σ .

Interestingly, it is not possible to apply the argument used for $\text{CC}^=$ to show that a CC^\sim compiler guarantees $\mathbf{W} \downarrow \models \tilde{\tau}(\mathbf{H}_S)$ whenever $\mathbf{W} \models \mathbf{H}_S$. This is in fact not true because direct images do not necessarily preserve subset-closure [37, 45]. To fix this we close the image of $\tilde{\tau}$ and $\tilde{\sigma}$ under subsets (denoted as Cl_\subseteq) and obtain:

Theorem 2.11 (Preservation of Subset-Closed Hyperproperties \clubsuit). For any trace relation \sim and its existential and universal images lifted to hyperproperties, $\tilde{\tau}$ and $\tilde{\sigma}$, and for $\text{Cl}_\subseteq(H) = \{\pi \mid \exists \pi' \in H. \pi \subseteq \pi'\}$, we have:

$$\begin{aligned} \text{SCHP}^{\text{Cl}_\subseteq \circ \tilde{\tau}} &\iff \text{CC}^\sim \iff \text{SCHP}^{\text{Cl}_\subseteq \circ \tilde{\sigma}}, \text{ where} \\ \text{SCHP}^{\text{Cl}_\subseteq \circ \tilde{\tau}} &\equiv \forall \mathbf{W} \forall \mathbf{H}_S \in \text{SCH}_S. \mathbf{W} \models \mathbf{H}_S \Rightarrow \mathbf{W} \downarrow \models \text{Cl}_\subseteq(\tilde{\tau}(\mathbf{H}_S)); \\ \text{SCHP}^{\text{Cl}_\subseteq \circ \tilde{\sigma}} &\equiv \forall \mathbf{W} \forall \mathbf{H}_T \in \text{SCH}_T. \mathbf{W} \models \text{Cl}_\subseteq(\tilde{\sigma}(\mathbf{H}_T)) \Rightarrow \mathbf{W} \downarrow \models \mathbf{H}_T. \end{aligned}$$

Theorem 2.11 makes us aware of the potential loss of precision when interested in preserving subset-closed hyperproperties through compilation. In §4 we focus on a security relevant subset-closed hyperproperty, noninterference, and show that such a loss of precision can be intended as a declassification of noninterference.

3 Instances of Trace-Relating Compiler Correctness

The trace-relating view of compiler correctness above can serve as a unifying framework for studying a range of interesting compilers. This section provides several representative instantiations of the framework: source languages with undefined behavior that compilation can turn into arbitrary target behavior (§3.1), target languages with resource exhaustion that cannot happen in the source (§3.2), changes in the representation of values (§3.3), and differences in the granularity of data and observable events (§3.4).

3.1 Undefined Behavior

We start by expanding upon the discussion of undefined behavior in §1. We first study the model of *CompCert*, where source and target alphabets are the same, including the event for undefined behavior. The trace relation weakens equality by allowing undefined behavior to be replaced with an arbitrary sequence of events.

Example 3.1 (CompCert-like Undefined Behavior Relation). Source and target traces are sequences of events drawn from Σ , where *Goes_wrong* $\in \Sigma$ is a terminal event that represents an undefined behavior. We then use the trace relation from the introduction:

$$s \sim t \quad \equiv \quad s = t \vee \exists m \leq t. s = m \cdot \text{Goes_wrong}.$$

Each trace of a target program produced by a CC^\sim compiler is either also a trace of the original source program or it has a finite prefix that the source program also produces, immediately before encountering undefined behavior. As explained in §1, one of the correctness theorems in *CompCert* can be rephrased as this variant of CC^\sim .

We proved that the property mappings induced by the relation can be written as (4):

$$\begin{aligned}\tilde{\sigma}(\pi_T) &= \{s \mid s \in \pi_T \wedge s \neq m \cdot \text{Goes_wrong}\} \cup \{m \cdot \text{Goes_wrong} \mid \forall t. m \leq t \implies t \in \pi_T\}; \\ \tilde{\tau}(\pi_S) &= \{t \mid t \in \pi_S\} \cup \{t \mid \exists m \leq t. m \cdot \text{Goes_wrong} \in \pi_S\}.\end{aligned}$$

These two mappings explain what a CC^\sim compiler ensures for the \sim relation above. The target-to-source mapping $\tilde{\sigma}$ states that to prove that a compiled program has a property π_T using source-level reasoning, one has to prove that any trace produced by the source program must either be a target trace satisfying π_T or have undefined behavior, but only provided that *any continuation* of the trace substituted for the undefined behavior satisfies π_T . The source-to-target mapping $\tilde{\tau}$ states that by compiling a program satisfying a property π_S we obtain a program that produces traces that satisfy the same property or that extend a source trace that ends in undefined behavior.

These definitions can help us reason about programs. For instance, $\tilde{\sigma}$ specifies that, to prove that an event does not happen in the target, it is not enough to prove that it does not happen in the source: it is also necessary to prove that the source program is does not have any undefined behavior (second disjunct). Indeed, if it had an undefined behavior, its continuations could exhibit the unwanted event. \square

This relation can be easily generalized to other settings. For instance, consider the setting in which we compile down to a low-level language like machine code. Target traces can now contain new events that cannot occur in the source: indeed, in modern architectures like x86 a compiler typically uses only a fraction of the available instruction set. Some instructions might even perform dangerous operations, such as writing to the hard drive. Formally, the source and target do not have the same events any more. Thus, we consider a source alphabet $\Sigma_S = \Sigma \cup \{\text{Goes_wrong}\}$, and a target alphabet $\Sigma_T = \Sigma \cup \Sigma'$. The trace relation is defined in the same way and we obtain the same property mappings as above, except that since target traces now have more events (some of which may be dangerous), and the arbitrary continuations of target traces get more interesting. For instance, consider a new event that represents writing data on the hard drive, and suppose we want to prove that this event cannot happen for a compiled program. Then, proving this property requires exactly proving that the source program exhibits no undefined behavior [11]. More generally, what one can prove about target-only events can only be either that they cannot appear (because there is no undefined behavior) or that any of them can appear (in the case of undefined behavior).

In §5.2 we study a similar example, showing that even in a safe language linked adversarial contexts can cause dangerous target events that have no source correspondent.

3.2 Resource Exhaustion

Let us return to the discussion about resource exhaustion in §1.

Example 3.2 (Resource Exhaustion). We consider traces made of events drawn from Σ_S in the source, and $\Sigma_T = \Sigma_S \cup \{\text{Resource_Limit_Hit}\}$ in the target. Recall the trace relation for resource exhaustion:

$$s \sim t \quad \equiv \quad s = t \vee \exists m \leq s. t = m \cdot \text{Resource_Limit_Hit}.$$

Formally, this relation is similar to the one for undefined behavior, except this time it is the target trace that is allowed to end early instead of the source trace.

The induced trace property mappings $\tilde{\sigma}$ and $\tilde{\tau}$ are the following (♣):

$$\tilde{\sigma}(\pi_T) = \{s \mid s \in \pi_T\} \cap \{s \mid \forall m \leq s. m \cdot \mathbf{Resource_Limit_Hit} \in \pi_T\};$$

$$\tilde{\tau}(\pi_S) = \pi_S \cup \{m \cdot \mathbf{Resource_Limit_Hit} \mid \exists s \in \pi_S. m \leq s\}.$$

These capture the following intuitions. The target-to-source mapping $\tilde{\sigma}$ states that to prove a property of the compiled program one has to show that the traces of the source program satisfy two conditions: (1) they must also satisfy the target property; and (2) the termination of every one of their prefixes by a resource exhaustion error must be allowed by the target property. This is rather restrictive: any property that prevents resource exhaustion cannot be proved using source-level reasoning. Indeed, if π_T does not allow resource exhaustion, then $\tilde{\sigma}(\pi_T) = \emptyset$. This is to be expected since resource exhaustion is simply not accounted for at the source level. The other mapping $\tilde{\tau}$ states that a compiled program produces traces that either belong to the same properties as the traces of the source program or end early due to resource exhaustion.

In this example, safety properties [32] are mapped (in both directions) to other safety properties (♣). This can be desirable for a relation: since safety properties are usually easier to reason about, one interested only in safety properties at the target can reason about them using source-level reasoning tools for safety properties. §B.1 provides a detailed account of safety properties and how one can restrict the notion of compiler correctness if one is only interested in them.

The compiler correctness theorem in CakeML is an instance of CC^\sim for the \sim relation above. We have also implemented two small compilers that are correct for this relation. The full details can be found in the Coq development. The first compiler (♣) goes from a simple expression language (similar to the one in §3.3 but without inputs) to the same language except that execution is bounded by some amount of fuel: each execution step consumes some amount of fuel and execution immediately halts when it runs out of fuel. The compiler is the identity.

The second compiler (♣) is more interesting: we proved this CC^\sim instance for a variant of a compiler from a WHILE language to a simple stack machine by Xavier Leroy [36]. We enriched the two languages with outputs and modified the semantics of the stack machine so that it falls into an error state if the stack reaches a certain size. The proof uses a standard forward simulation modified to account for failure. \square

We conclude this subsection by noting that the resource exhaustion relation and the undefined behavior relation from the previous subsection can easily be combined. Indeed, given a relation \sim_{UB} and a relation \sim_{RE} defined as above on the same sets of traces, we can build a new relation \sim that allows both refinement of undefined behavior and resource exhaustion by taking their union: $s \sim t \equiv s \sim_{UB} t \vee s \sim_{RE} t$. A compiler that is $CC^{\sim_{UB}}$ or $CC^{\sim_{RE}}$ is trivially CC^\sim , though the converse is not true.

3.3 Different Source and Target Values

We now illustrate trace-relating compilation for a translation mapping source-level booleans to target-level natural numbers. Given the simplicity of this compiler, most of the details of the formalization are deferred to the online appendix.

The source language is a pure, statically typed expression language whose expressions e include naturals n , booleans b , conditionals, arithmetic and relational operations,

boolean inputs in_b and natural inputs in_n . A trace s is a list of inputs is paired with a result r , which can be a natural, a boolean, or an error. Well-typed programs never produce error ($\not\Downarrow$). Types ty are either \mathbf{N} (naturals) or \mathbf{B} (booleans); typing is standard. The source language has a standard big-step operational semantics ($e \rightsquigarrow \langle \text{is}, r \rangle$) which tells how an expression e generates a trace $\langle \text{is}, r \rangle$. The target language is analogous, except that it is untyped, only has naturals \mathbf{n} and its only inputs are naturals in_n . The semantics of the target language is also given in big-step style. Since we only have naturals and all expressions operate on them, no error result is possible in the target.

The compiler is homomorphic, translating a source expression to the same target expression; the only differences are natural numbers (and conditionals), as noted below.

$$\begin{aligned} \text{true}\downarrow &= \mathbf{1} & \text{in}_b\downarrow &= \text{in}_n & e_1 \leq e_2\downarrow &= \text{if } e_1\downarrow \leq e_2\downarrow \text{ then } \mathbf{1} \text{ else } \mathbf{0} \\ \text{false}\downarrow &= \mathbf{0} & \text{in}_n\downarrow &= \text{in}_n & \text{if } e_1 \text{ then } e_2 \text{ else } e_3\downarrow &= \text{if } e_1\downarrow \leq \mathbf{0} \text{ then } e_3\downarrow \text{ else } e_2\downarrow \end{aligned}$$

When compiling an *if-then-else* the target condition $e_1\downarrow \leq \mathbf{0}$ is used to check that e_1 is false, and therefore the *then* and *else* branches of the source are swapped in the target.

Relating Traces. We relate basic values (naturals and booleans) in a non-injective fashion as noted below. Then, we extend the relation to lists of inputs pointwise (Rules **Empty** and **Cons**) and lift that relation to traces (Rules **Nat** and **Bool**).

$$\begin{array}{c} \begin{array}{ccc} n \sim n & \text{true} \sim n \text{ if } n > 0 & \text{false} \sim 0 \end{array} \\ \hline \begin{array}{ccc} \text{(Empty)} & \text{(Cons)} & \begin{array}{cc} \text{(Nat)} & \text{(Bool)} \\ \frac{\text{is} \sim \text{is} \quad n \sim n}{\langle \text{is}, n \rangle \sim \langle \text{is}, n \rangle} & \frac{\text{is} \sim \text{is} \quad b \sim n}{\langle \text{is}, b \rangle \sim \langle \text{is}, n \rangle} \end{array} \end{array} \\ \frac{\emptyset \sim \emptyset}{\frac{i \sim i \quad \text{is} \sim \text{is}}{i \cdot \text{is} \sim i \cdot \text{is}}} \end{array}$$

Property mappings. The property mappings $\tilde{\sigma}$ and $\tilde{\tau}$ induced by the trace relation \sim defined above capture the intuition behind encoding booleans as naturals:

- the source-to-target mapping allows **true** to be encoded by any non-zero number;
- the target-to-source mapping requires that **0** be replaceable by *both* **0** and **false**.

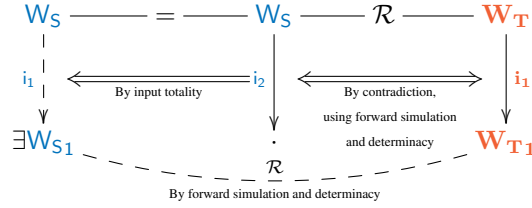
Compiler correctness. With the relation above, the compiler is proven to satisfy $\text{CC}\sim$.

Theorem 3.3 ($\cdot\downarrow$ is correct \Downarrow). $\cdot\downarrow$ is $\text{CC}\sim$.

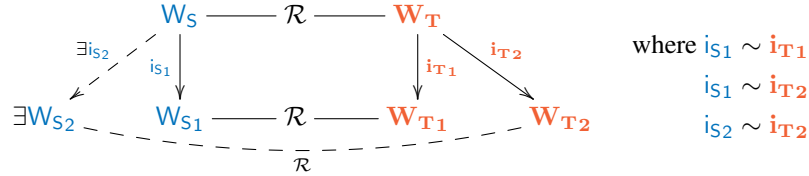
Simulations with different traces. The difficulty in proving **Theorem 3.3** arises from the trace-relating compilation setting: For compilation chains that have the same source and target traces, it is customary to prove compiler correctness using a forward simulation (i.e., a simulation between source and target transition system); then, using determinacy [18, 40] of the target language and input totality [19, 69] (aka receptiveness) of the source, this forward simulation is flipped into a backward simulation (a simulation between target and source transition system), as described by Beringer et al. [7], Leroy [35]. This flipping is useful because forward simulations are often much easier to prove (by induction on the transitions of the source) than backward ones, as it is the case here.

We first give the main idea of the flipping proof, when the inputs are the same in the source and the target [7, 35]. We only consider inputs, as it is the most interesting case, since with determinacy, nondeterminism only occurs on inputs. Given a forward simulation \mathcal{R} , and a target program \mathbf{W}_T that simulates a source program \mathbf{W}_S , \mathbf{W}_T is able to perform an input iff so is \mathbf{W}_S : otherwise, say for instance that \mathbf{W}_S performs an

output, by forward simulation \mathbf{W}_T would also perform an output, which is impossible because of determinacy. By input totality of the source, \mathbf{W}_S must be able to perform the exact same input as \mathbf{W}_T ; using forward simulation and determinacy, the resulting programs must be related.



However, our trace relation is not injective (both $\mathbf{0}$ and \mathbf{false} are mapped to $\mathbf{0}$), therefore these arguments do not apply: not all possible inputs of target programs are accounted for in the forward simulation. We thus have to strengthen the forward simulation assumption, requiring the following additional property to hold, for any source program \mathbf{W}_S and target program \mathbf{W}_T related by the forward simulation \mathcal{R} .



We say that a forward simulation for which this property holds is *flippable*. For our example compiler, a flippable forward simulation works as follows: whenever a boolean input occurs in the source, the target program must perform every strictly positive input \mathbf{n} (and not just $\mathbf{1}$, as suggested by the compiler). Using this property, determinacy of the target, input totality of the source, as well as the fact that any target input has an inverse image through the relation, we can indeed show that the forward simulation can be turned into a backward one: starting from $\mathbf{W}_S \mathcal{R} \mathbf{W}_T$ and an input \mathbf{i}_{T2} , we show that there is \mathbf{i}_{S1} and \mathbf{i}_{T2} as in the diagram above, using the same arguments as when the inputs are the same; because the simulation is flippable, we can close the diagram, and obtain the existence of an adequate \mathbf{i}_{S2} . From this we obtain CC^\sim .

In fact, we have proven a completely general ‘flipping theorem’, with this flippable hypothesis on the forward simulation (\mathcal{R}). We have also shown that if the relation \sim defines a bijection between the inputs of the source and the target, then any forward simulation is flippable, hence reobtaining the usual proof technique [7, 35] as a special case. This flipping theorem is further discussed in the online appendix.

3.4 Abstraction Mismatches

We now consider how to relate traces where a single source action is compiled to multiple target ones. To illustrate this, we take a pure, statically-typed source language that can output (nested) pairs of arbitrary size, and a pure, *untyped* target language where sent values have a fixed size. Concretely, the source is analogous to the language of §3.3, except that it does not have inputs or booleans and it has an expression $\mathbf{send\ e}$, which can emit a (nested) pair \mathbf{e} of values in a single action. That is, given that \mathbf{e} reduces to a pair, e.g., $\langle \mathbf{v1}, \langle \mathbf{v2}, \mathbf{v3} \rangle \rangle$, expression $\mathbf{send\ \langle v1, \langle v2, v3 \rangle \rangle}$ emits action $\langle \mathbf{v1}, \langle \mathbf{v2}, \mathbf{v3} \rangle \rangle$.

That expression is compiled into a sequence of individual sends in the target language $\text{send } v1 ; \text{send } v2 ; \text{send } v3$, since in the target, $\text{send } e$ sends the value that e reduces to, but the language has no pairs.

Due to space constraints we omit the full formalization of these simple languages and of the homomorphic compiler $((\cdot)\downarrow : e \rightarrow e)$. The only interesting bit is the compilation of the send expression, which relies on the $\text{gensend}(\cdot)$ function below. That function takes a source expression of a given type and returns a sequence of target send instructions that send each element of the expression.

$$\text{gensend}(\vdash e : \tau) = \begin{cases} \text{send}(\vdash e : N)\downarrow & \text{if } \tau = N \\ \text{gensend}(\vdash e.1 : \tau'); \text{gensend}(\vdash e.2 : \tau'') & \text{if } \tau = \tau' \times \tau'' \end{cases}$$

Relating Traces. We start with the trivial relation between numbers: $n \sim^0 n$, i.e., numbers are related when they are the same. We cannot build a relation between single actions since a single source action is related to multiple target ones. Therefore, we define a relation between a source action M and a target trace t (a list of numbers), inductively on the structure of M (which is a pair of values, and values are natural numbers or pairs).

$$\begin{array}{c} \text{(Trace-Rel-N-N)} \\ \frac{n \sim^0 n \quad n' \sim^0 n'}{\langle n, n' \rangle \sim n \cdot n'} \end{array} \quad \begin{array}{c} \text{(Trace-Rel-N-M)} \\ \frac{n \sim^0 n \quad M \sim t}{\langle n, M \rangle \sim n \cdot t} \end{array} \quad \begin{array}{c} \text{(Trace-Rel-M-N)} \\ \frac{M \sim t \quad n \sim^0 n}{\langle M, n \rangle \sim t \cdot n} \end{array} \quad \begin{array}{c} \text{(Trace-Rel-M-M)} \\ \frac{M \sim t \quad M' \sim t'}{\langle M, M' \rangle \sim t \cdot t'} \end{array}$$

A pair of naturals is related to the two actions that send each element of the pair (Rule **Trace-Rel-N-N**). If a pair is made of sub-pairs, we require all such sub-pairs to be related (Rules **Trace-Rel-N-M** to **Trace-Rel-M-M**). We build on these rules to define the $s \sim t$ relation between source and target traces for which the compiler is correct (Theorem 3.4). Trivially, traces are related when they are both empty. Alternatively, given related traces, we can concatenate a source action and a second target trace provided that they are related (Rule **Trace-Rel-Single**).

$$\begin{array}{c} \text{(Trace-Rel-Single)} \\ \frac{s \sim t \quad M \sim t'}{s \cdot M \sim t \cdot t'} \end{array}$$

Theorem 3.4 $((\cdot)\downarrow$ is correct). $(\cdot)\downarrow$ is $CC\sim$.

With our trace relation, the trace property mappings capture the following intuitions:

- The target-to-source mapping states that a source property can reconstruct target action as it sees fit. For example, trace $4 \cdot 6 \cdot 5 \cdot 7$ is related to $\langle 4, 6 \rangle \cdot \langle 5, 7 \rangle$ and $\langle\langle 4, \langle 6, \langle 5, 7 \rangle \rangle \rangle\rangle$ (and many more variations). This gives freedom to the source implementation of a target behavior, which follows from the non-injectivity of \sim .⁵
- The source-to-target mapping “forgets” about the way pairs are nested, but is faithful w.r.t. the values v_i contained in a message. Notice that source safety properties are always mapped to target safety properties. For instance, if $\pi_S \in \text{Safety}_S$ prescribes that some bad number is never sent, then $\tilde{\tau}(\pi_S)$ prescribes the same number is never sent in the target and $\tilde{\tau}(\pi_S) \in \text{Safety}_T$. Of course if $\pi_S \in \text{Safety}_S$ prescribes that a particular nested pairing like $\langle 4, \langle 6, \langle 5, 7 \rangle \rangle$ never happens, then $\tilde{\tau}(\pi_S)$ is still a target safety property, but the trivial one, since $\tilde{\tau}(\pi_S) = \top \in \text{Safety}_T$.

⁵ Making \sim injective is a matter of adding open and close parenthesis actions in target traces.

4 Trace-Relating Compilation and Noninterference Preservation

When source and target observations are drawn from the same set, a correct compiler ($CC^=$) is enough to ensure the preservation of all subset-closed hyperproperties, in particular of *noninterference* (NI) [22], as also mentioned at the beginning of §2.3. In the scenario where target observations are strictly more informative than source observations, the best guarantee one may expect from a correct trace-relating compiler (CC^\sim) is a *weakening* (or *declassification*) of target noninterference that matches the noninterference property satisfied in the source. To formalize this reasoning, this section applies the trinitarian view of trace-relating compilation to the general framework of abstract noninterference (ANI) [21].

We first define NI and explain the issue of preserving source NI via a CC^\sim compiler. We then introduce ANI, which allows characterizations of various forms of noninterference, and formulate a general theory of ANI preservation via CC^\sim . We also study how to deal with cases such as undefined behavior in the target. Finally, we answer the dual question, i.e., which source NI should be satisfied to guarantee that compiled programs are noninterfering with respect to target observers.

Intuitively, NI requires that publicly observable outputs do not reveal information about private inputs. To define this formally, we need a few additions to our setup. We indicate the (disjoint) *input* and *output* projections of a trace t as t° and t^\star respectively⁶. Denote with $[t]_{low}$ the equivalence class of a trace t , obtained using a standard low-equivalence relation that relates low (public) events only if they are equal, and ignores any difference between private events. Then, NI for source traces can be defined as:

$$NI_S = \{ \pi_S \mid \forall s_1 s_2 \in \pi_S. [s_1^\circ]_{low} = [s_2^\circ]_{low} \Rightarrow [s_1^\star]_{low} = [s_2^\star]_{low} \}.$$

That is, source NI comprises the sets of traces that have equivalent low output projections as long as their low input projections are equivalent.

Trace-Relating Compilation and Noninterference. When additional observations are possible in the target, it is unclear whether a noninterfering source program is compiled to a noninterfering target program or not, and if so, whether the notion of NI in the target is the expected or desired one. We illustrate this issue considering a scenario where target traces extend source ones by exposing the execution time. While source noninterference NI_S requires that private inputs do not affect public outputs, NI_T additionally requires that the execution time is not affected by private inputs.

To model the scenario described, let $Trace_S$ denote the set of traces in the source, and $Trace_T = Trace_S \times \mathbb{N}^\omega$ be the set of target traces, where $\mathbb{N}^\omega \triangleq \mathbb{N} \cup \{\omega\}$. Target traces have two components: a source trace, and a natural number that denotes the time spent to produce the trace (ω if infinite). Notice that if two source traces s_1, s_2 , are low-equivalent then $\{s_1, s_2\} \in NI_S$ and $\{(s_1, 42), (s_1, 42)\} \in NI_T$, but $\{(s_1, 42), (s_2, 43)\} \notin NI_T$ and $\{(s_1, 42), (s_2, 42), (s_1, 43), (s_2, 43)\} \notin NI_T$.

Consider the following straightforward trace relation, which relates a source trace to any target trace whose first component is equal to it, irrespective of execution time:

$$s \sim t \quad \equiv \quad \exists n. t = (s, n).$$

⁶ Here we only require the projections to be disjoint. Depending on the scenario and the attacker model the projections might record information such as the ordering of events.

A compiler is CC^\sim if any trace that can be exhibited in the target can be simulated in the source in some amount of time. For such a compiler [Theorem 2.11](#) says that if W satisfies NI_S , then $W\downarrow$ satisfies $Cl_{\subseteq} \circ \tilde{\tau}(NI_S)$, which however is strictly weaker than NI_T , as it contains, e.g., $\{(s_1, 42), (s_2, 42), (s_1, 43), (s_2, 43)\}$, and one cannot conclude that $W\downarrow$ is noninterfering in the target. It is easy to prove that

$$Cl_{\subseteq} \circ \tilde{\tau}(NI_S) = Cl_{\subseteq} (\{ \pi_S \times \mathbb{N}^\omega \mid \pi_S \in NI_S \}) = \{ \pi_S \times \mathcal{I} \mid \pi_S \in NI_S \wedge \mathcal{I} \subseteq \mathbb{N}^\omega \},$$

the first equality coming from $\tilde{\tau}(\pi_S) = \pi_S \times \mathbb{N}^\omega$, and the second from NI_S being subset-closed. As we will see, this hyperproperty *can* be characterized as a form of NI, which one might call *timing-insensitive noninterference*, and ensured only against attackers that cannot measure execution time. For this characterization, and to describe different forms of noninterference as well as formally analyze their preservation by a CC^\sim compiler, we rely on the general framework of *abstract noninterference* [21].

Abstract Noninterference. ANI [21] is a generalization of NI whose formulation relies on abstractions (in abstract interpretation sense [16]) in order to encompass arbitrary variants of NI. ANI is parameterized by an *observer abstraction* ρ , which denotes the distinguishing power of the attacker, and a *selection abstraction* ϕ , which specifies when to check NI, and therefore captures a form of declassification [58].⁷ Formally:

$$ANI_{\phi}^{\rho} = \{ \pi \mid \forall t_1 t_2 \in \pi. \phi(t_1) = \phi(t_2) \Rightarrow \rho(t_1) = \rho(t_2) \}.$$

By picking $\phi = \rho = [\cdot]_{low}$, we recover the standard noninterference defined above, where NI must hold for all low inputs (i.e., no declassification of private inputs), and the observational power of the attacker is limited to distinguishing low outputs.

The observational power of the attacker can be weakened by choosing a more liberal relation for ρ . For instance, one may limit the attacker to observe the *parity* of output integer values. Another way to weaken ANI is to use ϕ to specify that noninterference is only required to hold for a subset of low inputs.

To be formally precise, ϕ and ρ are defined over sets of (input and output projections of) traces, so when we write $\phi(t)$ above, this should be understood as a convenience notation for $\phi(\{t\})$. Likewise, $\phi = [\cdot]_{low}$ should be understood as $\phi = \lambda\pi. \bigcup_{t \in \pi} [t]_{low}$, i.e., the powerset lifting of $[\cdot]_{low}$. Additionally, ϕ and ρ are required to be upper-closed operators (*uco*)—i.e., monotonic, idempotent and extensive—on the poset that is the powerset of (input and output projections of) traces ordered by inclusion [21].

Trace-Relating Compilation and ANI for Timing. We can now reformulate our example with observable execution times in the target in terms of ANI. We have $NI_S = ANI_{\rho_S}^{\phi_S}$ with $\phi_S = \rho_S = [\cdot]_{low}$. In this case, we can formally describe the hyperproperty that a compiled program $W\downarrow$ satisfies whenever W satisfies NI_S as an instance of ANI:

$$Cl_{\subseteq} \circ \tilde{\tau}(NI_S) = ANI_{\phi_T}^{\rho_T},$$

$$\text{for } \phi_T = \phi_S \text{ and } \rho_T(\pi_T) = \{ (s, n) \mid \exists (s_1, n_1) \in \pi_T. [s^*]_{low} = [s_1^*]_{low} \}.$$

The definition of ϕ_T tells us that the trace relation does not affect the selection abstraction. The definition of ρ_T characterizes an observer that cannot distinguish execution times for noninterfering traces (notice that n_1 in the definition of ρ_T is discarded). For instance, $\rho_T(\{(s, n_1)\}) = \rho_T(\{(s, n_2)\})$, for any s, n_1, n_2 . Therefore, in this setting,

⁷ ANI includes a third parameter η , which describes the maximal input variation that the attacker may control. Here we omit η (i.e., take it to be the identity) in order to simplify the presentation.

we know explicitly through $\rho_{\mathbf{T}}$ that a CC^{\sim} compiler degrades source noninterference to target *timing-insensitive* noninterference.

Trace-Relating Compilation and ANI in General. While the particular $\phi_{\mathbf{T}}$ and $\rho_{\mathbf{T}}$ above can be discovered by intuition, we want to know whether there is a systematic way of obtaining them in general. In other words, for *any* trace relation \sim and *any* notion of source NI, what property is guaranteed on noninterfering source programs by any CC^{\sim} compiler?

We can now answer this question generally (Theorem 4.1): any source notion of noninterference expressible as an instance of ANI is mapped to a corresponding instance of ANI in the target, whenever source traces are an abstraction of target ones (i.e., when \sim is a total and surjective map). For this result we consider trace relations that can be split into input and output trace relations (denoted as $\sim \triangleq \langle \tilde{\sim}, \tilde{\sim} \rangle$) such that $s \sim t \iff s^\circ \tilde{\sim} t^\circ \wedge s^\bullet \tilde{\sim} t^\bullet$. The trace relation \sim corresponds to a Galois connection between the sets of trace properties $\tilde{\tau} \sqsubseteq \tilde{\sigma}$ as described in §2.2. Similarly, the pair $\tilde{\sim}$ and $\tilde{\sim}$ corresponds to a pair of Galois connections, $\tilde{\tau}^\circ \sqsubseteq \tilde{\sigma}^\circ$ and $\tilde{\tau}^\bullet \sqsubseteq \tilde{\sigma}^\bullet$, between the sets of input and output properties. In the timing example, time is an output so we have $\sim \triangleq \langle =, \tilde{\sim} \rangle$ and $\tilde{\sim}$ is defined as $s^\bullet \tilde{\sim} t^\bullet \equiv \exists n. t^\bullet = (s^\bullet, n)$.

Theorem 4.1 (Compiling ANI). *Assume traces of source and target languages are related via $\sim \subseteq \text{Traces}_S \times \text{Trace}_T$, $\sim \triangleq \langle \tilde{\sim}, \tilde{\sim} \rangle$ such that $\tilde{\sim}$ and $\tilde{\sim}$ are both total maps from target to source traces, and $\tilde{\sim}$ is surjective. Assume \downarrow is a CC^{\sim} compiler, and $\phi_S \in \text{uco}(2^{\text{Trace}_S^\circ})$, $\rho_S \in \text{uco}(2^{\text{Trace}_S^\bullet})$.*

If \mathbb{W} satisfies $\text{ANI}_{\phi_S}^{\rho_S}$, then $\mathbb{W}\downarrow$ satisfies $\text{ANI}_{\phi_{\mathbf{T}}^\#}^{\rho_{\mathbf{T}}^\#}$, where $\phi_{\mathbf{T}}^\#$ and $\rho_{\mathbf{T}}^\#$ are defined as:

$$\begin{aligned} \phi_{\mathbf{T}}^\# &= g^\circ \circ \phi_S \circ f^\circ; & \rho_{\mathbf{T}}^\# &= g^\bullet \circ \rho_S \circ f^\bullet \quad \text{and} \\ f^\circ(\pi_{\mathbf{T}}^\circ) &= \{s^\circ \mid \exists t^\circ \in \pi_{\mathbf{T}}^\circ. s^\circ \tilde{\sim} t^\circ\}; & g^\circ(\pi_S^\circ) &= \{t^\circ \mid \forall s^\circ. s^\circ \tilde{\sim} t^\circ \Rightarrow s^\circ \in \pi_S^\circ\} \end{aligned}$$

(and both f^\bullet and g^\bullet are defined analogously).

For the example above we recover the definitions we justified intuitively, i.e., $\phi_{\mathbf{T}}^\# = g^\circ \circ \phi_S \circ f^\circ = \phi_{\mathbf{T}}$ and $\rho_{\mathbf{T}}^\# = g^\bullet \circ \rho_S \circ f^\bullet = \rho_{\mathbf{T}}$. Moreover, we can prove that if $\tilde{\sim}$ also is surjective, $\text{ANI}_{\phi_{\mathbf{T}}^\#}^{\rho_{\mathbf{T}}^\#} \subseteq \text{Cl}_{\subseteq} \circ \tilde{\tau}(\text{ANI}_{\phi_S}^{\rho_S})$. Therefore, the derived guarantee $\text{ANI}_{\phi_{\mathbf{T}}^\#}^{\rho_{\mathbf{T}}^\#}$ is at least as strong as the one that follows by just knowing that the compiler \downarrow is CC^{\sim} .

Noninterference and Undefined Behavior. As stated above, Theorem 4.1 does not apply to several scenarios from §3 such as undefined behavior (§3.1), as in those cases the relation $\tilde{\sim}$ is not a total map. Nevertheless, we can still exploit our framework to reason about the impact of compilation on noninterference.

Let us consider $\sim \triangleq \langle \tilde{\sim}, \tilde{\sim} \rangle$ where $\tilde{\sim}$ is any total and surjective map from target to source inputs (e.g., equality) and $\tilde{\sim}$ is defined as $s^\bullet \tilde{\sim} t^\bullet \equiv s^\bullet = t^\bullet \vee \exists m^\bullet \leq t^\bullet. s^\bullet = m^\bullet \cdot \text{Goes_wrong}$. Intuitively, a CC^{\sim} compiler guarantees that no interference can be observed by a target attacker that cannot exploit undefined behavior to learn private information. This intuition can be made formal by the following theorem.

Theorem 4.2 (Relaxed Compiling ANI). *Relax the assumptions of Theorem 4.1 by allowing $\tilde{\sim}$ to be any output trace relation. If \mathbb{W} satisfies $\text{ANI}_{\phi_S}^{\rho_S}$, then $\mathbb{W}\downarrow$ satisfies*

$ANI_{\phi_{\mathbb{T}}^{\#}}^{\rho_{\mathbb{T}}^{\#}}$ where $\phi_{\mathbb{T}}^{\#}$ is defined as in Theorem 4.1, and $\rho_{\mathbb{T}}^{\#}$ is such that:

$$\forall s \ t. s^{\circ} \sim t^{\circ} \Rightarrow \rho_{\mathbb{T}}^{\#}(t^{\circ}) = \rho_{\mathbb{T}}^{\#}(\tilde{\tau}^{\circ}(\rho_S(s^{\circ}))).$$

Technically, instead of giving us a *definition* of $\rho_{\mathbb{T}}^{\#}$, the theorem gives a *property* of it. The property states that, given a target output trace t° , the attacker cannot distinguish it from any other target output traces produced by other possible compilations ($\tilde{\tau}^{\circ}$) of the source trace s it relates to, up to the observational power of the source level attacker ρ_S . Therefore, given a source attacker ρ_S , the theorem characterizes a *family* of attackers that cannot observe any interference for a correctly compiled noninterfering program. Notice that the target attacker $\rho_{\mathbb{T}}^{\#} = \lambda_{\cdot}$. \top satisfies the premise of the theorem, but defines a trivial hyperproperty, so that we cannot prove in general that $ANI_{\phi_{\mathbb{T}}^{\#}}^{\rho_{\mathbb{T}}^{\#}} \subseteq Cl_{\subseteq} \circ \tilde{\tau}(ANI_{\phi_S}^{\rho_S})$. The same $\rho_{\mathbb{T}}^{\#} = \lambda_{\cdot}$. \top shows that the family of attackers described in Theorem 4.2 is nonempty, and this ensures the existence of a most powerful attacker among them [21], whose explicit characterization we leave for future work.

From Target NI to Source NI. We now explore the dual question: under what hypotheses does trace-relating compiler correctness alone allow target noninterference to be reduced to source noninterference? This is of practical interest, as one would be able to protect from target attackers by ensuring noninterference in the source. This task can be made easier if the source language has some static enforcement mechanism [1, 37].

Let us consider the languages from §3.4 extended with inputting of (pairs of) values. It is easy to show that the compiler described in §3.4 is still CC^{\sim} . Assume that we want to satisfy a given notion of target noninterference after compilation, i.e., $W \downarrow \models ANI_{\phi_{\mathbb{T}}}^{\rho_{\mathbb{T}}}$. Recall that the observational power of the target attacker, $\rho_{\mathbb{T}}$, is expressed as a property of sequences of values. To express the same property (or attacker) in the source, we have to abstract the way pairs of values are nested. For instance, the source attacker should not distinguish $\langle v_1, \langle v_2, v_3 \rangle \rangle$ and $\langle \langle v_1, v_2 \rangle, v_3 \rangle$. In general (i.e., when \sim is not the identity), this argument is valid only when $\phi_{\mathbb{T}}$ can be represented in the source. More precisely, $\phi_{\mathbb{T}}$ must consider as equivalent all target inputs that are related to the same source one, because in the source it is not possible to have a finer distinction of inputs. This intuitive correspondence can be formalized as follows:

Theorem 4.3 (Target ANI by source ANI). Let $\phi_{\mathbb{T}} \in uco(2^{\text{Trace}_{\mathbb{T}}^{\circ}})$, $\rho_{\mathbb{T}} \in uco(2^{\text{Trace}_{\mathbb{T}}^{\circ}})$ and \sim a total and surjective map from source outputs to target ones and assume that

$$\forall s \ t. s^{\circ} \sim t^{\circ} \Rightarrow \phi_{\mathbb{T}}(t^{\circ}) = \phi_{\mathbb{T}}(\tilde{\tau}^{\circ}(s^{\circ})).$$

If $\cdot \downarrow$ is a CC^{\sim} compiler and W satisfies $ANI_{\phi_S}^{\rho_S}$, then $W \downarrow$ satisfies $ANI_{\phi_{\mathbb{T}}}^{\rho_{\mathbb{T}}}$ for

$$\phi_S^{\#} = \tilde{\sigma}^{\circ} \circ \phi_{\mathbb{T}} \circ \tilde{\tau}^{\circ}; \quad \rho_S^{\#} = \tilde{\sigma}^{\circ} \circ \rho_{\mathbb{T}} \circ \tilde{\tau}^{\circ}.$$

To wrap up the discussion about noninterference, the results presented in this section formalize and generalize some intuitive facts about compiler correctness and noninterference. Of course, they all place some restrictions on the shape of the noninterference instances that can be considered, because compiler correctness alone is in general not a strong enough criterion for dealing with many security properties [6, 17].

5 Trace-Relating Secure Compilation

So far we have studied compiler correctness criteria for whole, standalone programs. However, in practice, programs do not exist in isolation, but in a context where they interact with other programs, libraries, etc. In many cases, this context cannot be assumed to be benign and could instead behave maliciously to try to disrupt a compiled program.

Hence, in this section we consider the following *secure compilation* scenario: a source program is compiled and linked with an arbitrary target-level context, i.e., one that may not be expressible as the compilation of a source context. Compiler correctness does not address this case, as it does not consider arbitrary target contexts, looking instead at whole programs (empty context [34]) or well-behaved target contexts that behave like source ones (as in compositional compiler correctness [27, 31, 46, 63]).

To account for this scenario, Abate et al. [2] describe several secure compilation criteria based on the preservation of classes of (hyper)properties (e.g., trace properties, safety, hypersafety, hyperproperties, etc.) against arbitrary target contexts. For each of these criteria, they give an equivalent “property-free” criterion, analogous to the equivalence between TP and CC^\equiv . For instance, their *robust* trace property preservation criterion (RTP) states that, for any trace property π , if a source *partial* program P plugged into any context C_S satisfies π , then the compiled program $P\downarrow$ plugged into any target context C_T satisfies π . Their equivalent criterion to RTP is RTC, which states that for any trace produced by the compiled program, when linked with any target context, there is a source context that produces the same trace. Formally (writing $C[P]$ to mean the whole program that results from linking partial program P with context C) they define:

$$\begin{aligned} \text{RTP} &\equiv \forall P. \forall \pi. (\forall C_S. \forall t. C_S[P] \rightsquigarrow t \Rightarrow t \in \pi) \Rightarrow (\forall C_T. \forall t. C_T[P\downarrow] \rightsquigarrow t \Rightarrow t \in \pi); \\ \text{RTC} &\equiv \forall P. \forall C_T. \forall t. C_T[P\downarrow] \rightsquigarrow t \Rightarrow \exists C_S. C_S[P] \rightsquigarrow t. \end{aligned}$$

In the following we adopt the notation $P \models_R \pi$ to mean “ P robustly satisfies π ,” i.e., P satisfies π irrespective of the contexts it is linked with. Thus, we write more compactly:

$$\text{RTP} \equiv \forall \pi. \forall P. P \models_R \pi \Rightarrow P\downarrow \models_R \pi.$$

All the criteria of Abate et al. [2] share this flavor of stating the existence of some source context that simulates the behavior of any given target context, with some variations depending on the class of (hyper)properties under consideration. All these criteria are stated in a setting where source and target traces are the same. In this section, we extend their result to our trace-relating setting, obtaining trintarian views for secure compilation. Despite the similarities with §2, more challenges show up, in particular when considering the robust preservation of proper sub-classes of trace properties. For example, after application of $\tilde{\sigma}$ or $\tilde{\tau}$, a property may not be safety anymore, a crucial point for the equivalence with the property-free criterion for safety properties by Abate et al. [2]. We solve this by interpreting the class of safety properties as an *abstraction* of the class of all trace properties induced by a closure operator (§5.1). The remaining subsections provide example compilation chains satisfying our trace-relating secure compilation criteria for trace properties (§5.2) and for safety properties hypersafety (§5.3).

5.1 Trace-Relating Secure Compilation: A Spectrum of Trinities

In this subsection we generalize many of the criteria of Abate et al. [2] using the ideas of §2. Before discussing how we solve the challenges for classes such as safety and

hypersafety, we show the simple generalization of RTC to the trace-relating setting (RTC \sim) and its corresponding trinitarian view (Theorem 5.1):

Theorem 5.1 (Trinity for Robust Trace Properties \clubsuit). *For any trace relation \sim and induced property mappings $\tilde{\tau}$ and $\tilde{\sigma}$, we have: $\text{RTP}^{\tilde{\tau}} \iff \text{RTC}^{\sim} \iff \text{RTP}^{\tilde{\sigma}}$, where*

$$\begin{aligned} \text{RTC}^{\sim} &\equiv \forall P \forall C_T \forall t. C_T [P \downarrow] \rightsquigarrow t \Rightarrow \exists C_S \exists s \sim t. C_S [P] \rightsquigarrow s; \\ \text{RTP}^{\tilde{\tau}} &\equiv \forall P \forall \pi_S \in 2^{\text{Traces}}. P \models_R \pi_S \Rightarrow P \downarrow \models_R \tilde{\tau}(\pi_S); \\ \text{RTP}^{\tilde{\sigma}} &\equiv \forall P \forall \pi_T \in 2^{\text{Traces}_T}. P \models_R \tilde{\sigma}(\pi_T) \Rightarrow P \downarrow \models_R \pi_T. \end{aligned}$$

Abate et al. [2] propose many more equivalent pairs of criteria, each preserving different classes of (hyper)properties, which we briefly recap now. For trace properties, they also have criteria that preserve safety properties plus their version of liveness properties. For hyperproperties, they have criteria that preserve hypersafety properties, subset-closed hyperproperties, and arbitrary hyperproperties. Finally, they define *relational* hyperproperties, which are relations between the behaviors of multiple programs for expressing, e.g., that a program always runs faster than another. For relational hyperproperties, they have criteria that preserve arbitrary relational properties, relational safety properties, relational hyperproperties and relational subset-closed hyperproperties. Roughly speaking, the security guarantees due to robust preservation of trace properties regard only protecting the integrity of the program from the context, the guarantees of hyperproperties also regard data confidentiality, and the guarantees of relational hyperproperties even regard code confidentiality. Naturally, these stronger guarantees are increasingly harder to enforce and prove.

While we have lifted the most significant criteria from Abate et al. [2] to our trinitarian view, due to space constraints we provide the formal definitions only for the two most interesting criteria. We summarize the generalizations of many other criteria in Figure 2, described at the end. Omitted definitions are available in the online appendix.

Beyond Trace Properties: Robust Safety and Hyperproperty Preservation. We detail robust preservation of safety properties and of arbitrary hyperproperties since they are both relevant from a security point of view and their generalization is interesting.

Theorem 5.2 (Trinity for Robust Safety Properties \clubsuit). *For any trace relation \sim and for the induced property mappings $\tilde{\tau}$ and $\tilde{\sigma}$, we have:*

$$\begin{aligned} \text{RTP}^{\text{Safe} \circ \tilde{\tau}} &\iff \text{RSC}^{\sim} \iff \text{RSP}^{\tilde{\sigma}}, & \text{where} \\ \text{RSC}^{\sim} &\equiv \forall P \forall C_T \forall t \forall m \leq t. C_T [P \downarrow] \rightsquigarrow t \Rightarrow \exists C_S \exists t' \geq m \exists s \sim t'. C_S [P] \rightsquigarrow s; \\ \text{RTP}^{\text{Safe} \circ \tilde{\tau}} &\equiv \forall P \forall \pi_S \in 2^{\text{Traces}}. P \models_R \pi_S \Rightarrow P \downarrow \models_R (\text{Safe} \circ \tilde{\tau})(\pi_S); \\ \text{RSP}^{\tilde{\sigma}} &\equiv \forall P \forall \pi_T \in \text{Safety}_T. P \models_R \tilde{\sigma}(\pi_T) \Rightarrow P \downarrow \models_R \pi_T. \end{aligned}$$

There is an interesting asymmetry between the last two characterizations above, which we explain now in more detail. $\text{RSP}^{\tilde{\sigma}}$ quantifies over target safety properties, while $\text{RTP}^{\text{Safe} \circ \tilde{\tau}}$ quantifies over *arbitrary* source properties, but imposes the composition of $\tilde{\tau}$ with *Safe*, which maps an arbitrary target property π_T to the target safety property that best over-approximates π_T ⁸ (an analogous *closure* was needed for subset-closed

⁸ $\text{Safe}(\pi_T) = \cap \{S_T \mid \pi_T \subseteq S_T \wedge S_T \in \text{Safety}_T\}$ is the topological closure in the topology of Clarkson and Schneider [14], where safety properties coincide with the closed sets.

hyperproperties in [Theorem 2.11](#)). More precisely, $Safe$ is a closure operator on target properties, with $\mathbf{Safety}_T = \{Safe(\pi_T) \mid \pi_T \in 2^{\mathbf{Trace}_T}\}$. The mappings

$$Safe \circ \tilde{\tau} : 2^{\mathbf{Traces}} \rightleftharpoons \mathbf{Safety}_T : \tilde{\sigma}$$

determine a Galois connection between source trace properties and target safety properties, and ensure the equivalence $\text{RTP}^{Safe \circ \tilde{\tau}} \iff \text{RSP}^{\tilde{\sigma}}$ (\clubsuit). This argument generalizes to arbitrary closure operators on target properties (\clubsuit) and on hyperproperties, as long as the corresponding class is a sub-class of subset-closed hyperproperties, and explains all but one of the asymmetries in [Figure 2](#), the one that concerns the robust preservation of arbitrary hyperproperties:

Theorem 5.3 (Weak Trinity for Robust Hyperproperties \clubsuit). *For a trace relation $\sim \subseteq \mathbf{Traces}_S \times \mathbf{Trace}_T$ and induced property mappings $\tilde{\sigma}$ and $\tilde{\tau}$, RHC^\sim is equivalent to $\text{RHP}^{\tilde{\tau}}$; moreover, if $\tilde{\tau} \rightleftharpoons \tilde{\sigma}$ is a Galois insertion (i.e., $\tilde{\tau} \circ \tilde{\sigma} = id$), RHC^\sim implies $\text{RHP}^{\tilde{\sigma}}$, while if $\tilde{\sigma} \rightleftharpoons \tilde{\tau}$ is a Galois reflection (i.e., $\tilde{\sigma} \circ \tilde{\tau} = id$), $\text{RHP}^{\tilde{\sigma}}$ implies RHC^\sim ,*

$$\text{where } \text{RHC}^\sim \equiv \forall P \forall C_T \exists C_S \forall t. C_T [P \downarrow] \rightsquigarrow t \iff (\exists s \sim t. C_S [P] \rightsquigarrow s);$$

$$\text{RHP}^{\tilde{\tau}} \equiv \forall P \forall H_S. P \models_R H_S \Rightarrow P \downarrow \models_R \tilde{\tau}(H_S);$$

$$\text{RHP}^{\tilde{\sigma}} \equiv \forall P \forall H_T. P \models_R \tilde{\sigma}(H_T) \Rightarrow P \downarrow \models_R H_T.$$

This trinity is *weak* since extra hypotheses are needed to prove some implications. While the equivalence $\text{RHC}^\sim \iff \text{RHP}^{\tilde{\tau}}$ holds unconditionally, the other two implications hold only under distinct, stronger assumptions. For $\text{RHP}^{\tilde{\sigma}}$ it is still possible and correct to deduce a source obligation for a given target hyperproperty H_T when no information is lost in the the composition $\tilde{\tau} \circ \tilde{\sigma}$ (i.e., the two maps are a Galois *insertion*). On the other hand, $\text{RHP}^{\tilde{\tau}}$ is a consequence of $\text{RHP}^{\tilde{\sigma}}$ when no information is lost in composing in the other direction, $\tilde{\sigma} \circ \tilde{\tau}$ (i.e., the two maps are a Galois *reflection*).

Navigating the Diagram. For a given trace relation \sim , [Figure 2](#) orders the generalized criteria according to their relative strength. If a trinity implies another (denoted by \Rightarrow), then the former provides stronger security for a compilation chain than the latter.

As mentioned, some property-full criteria regarding proper subclasses (i.e., subset-closed hyperproperties, safety, hypersafety, 2-relational safety and 2-relational hyperproperties) quantify over arbitrary (relational) (hyper)properties and compose $\tilde{\tau}$ with an additional operator. We have already presented the $Safe$ operator; other operators are Cl_C , $HSafe$, and $2rSafe$, which approximate the image of $\tilde{\tau}$ with a subset-closed hyperproperty, a hypersafety and 2-relational safety respectively.

As a reading aid, when quantifying over arbitrary trace properties we use the shaded blue as background color, we use the red when quantifying over arbitrary subset-closed hyperproperties and green for arbitrary 2-relational properties.

We now describe how to interpret the acronyms in [Figure 2](#). All criteria start with R meaning they refer to robust preservation. Criteria for relational hyperproperties—here only arity 2 is shown—contain 2r. Next, criteria names spell the class of hyperproperties they preserve: H for hyperproperties, SCH for subset-closed hyperproperties, HS for hypersafety, T for trace properties, and S for safety properties. Finally, property-free criteria end with a C while property-full ones involving $\tilde{\sigma}$ and $\tilde{\tau}$ end with P. Thus, *robust (R) subset-closed hyperproperty-preserving (SCH) compilation (C)* is RSCHC^\sim , *robust (R) two-relational (2r) safety-preserving (S) compilation (C)* is R2rSC^\sim , etc.

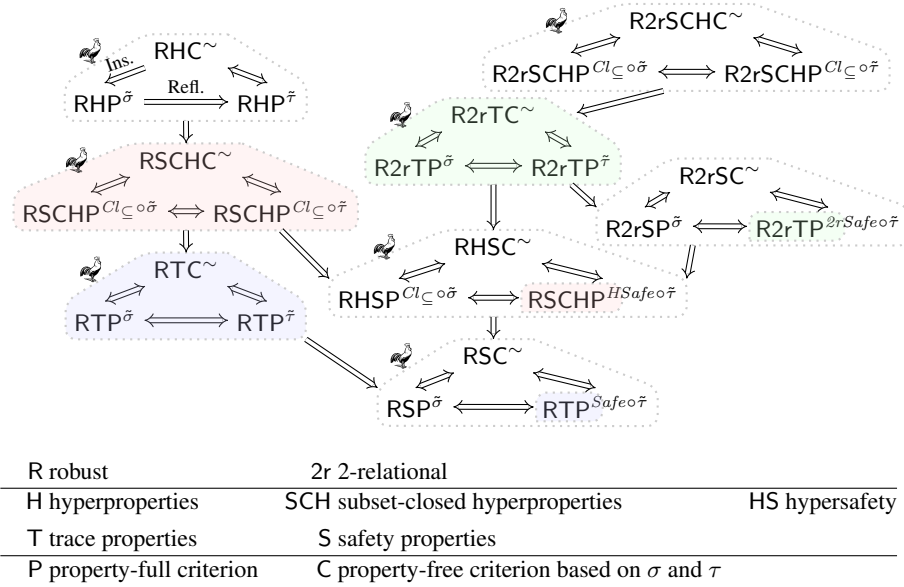


Fig. 2: Hierarchy of trinitarian views of secure compilation criteria preserving classes of hyperproperties and the key to read each acronym. Shorthands ‘Ins.’ and ‘Refl.’ stand for Galois Insertion and Reflection. The symbol denotes trinitities proven in Coq.

5.2 Instance of Trace-Relating Robust Preservation of Trace Properties

This subsection illustrates trace-relating secure compilation when the target language has strictly more events than the source that target contexts can exploit to break security.

Source and Target Languages. The source and target languages used here are nearly identical expression languages, borrowing from the syntax of the source language of §3.3. Both languages add *sequencing* of expressions, two kinds of *output events*, and the expressions that generate them: `outsn` and `outss` usable in source and target, respectively, and `outT` usable only in the target, which is the only difference between source and target. The extra events in the target model the fact that the target language has an increased ability to perform certain operations, some of them potentially dangerous (such as writing to the hard drive), which cannot be performed by the source language, and against which source-level reasoning can therefore offer no protection.

Both languages and compilation chains now deal with partial programs, contexts and linking of those two to produce whole programs. In this setting, a whole program is the combination of a *main expression* to be evaluated and a set of *function definitions* (with distinct names) that can refer to their argument symbolically and can be called by the main expression and by other functions. The set of functions of a whole program is the union of the functions of a partial program and a context; the latter also contains the main expression. The extensions of the typing rules and the operational semantics for whole programs are unsurprising and therefore elided. The trace model also follows closely that of §3.3: it consists of a list of *regular events* (including the new outputs)

terminated by a *result event*. Finally, a partial program and a context can be linked into a whole program when their functions satisfy the requirements mentioned above.

Relating Traces. In the present model, source and target traces differ only in the fact that the target draws (regular) events from a strictly larger set than the source, i.e., $\Sigma_T \supset \Sigma_S$. A natural relation between source and target traces essentially maps to a given target trace \mathbf{t} the source trace that erases from \mathbf{t} those events that exist only at the target level. Let $\mathbf{t}|_{\Sigma_S}$ indicate trace \mathbf{t} filtered to retain only those elements included in alphabet Σ_S . We define the trace relation as:

$$s \sim \mathbf{t} \quad \equiv \quad s = \mathbf{t}|_{\Sigma_S}.$$

In the opposite direction, a source trace s is related to many target ones, as any target-only events can be inserted at any point in s . The induced mappings for \sim are:

$$\tilde{\tau}(\pi_S) = \{\mathbf{t} \mid \exists s. s = \mathbf{t}|_{\Sigma_S} \wedge s \in \pi_S\}; \quad \tilde{\sigma}(\pi_T) = \{s \mid \forall \mathbf{t}. s = \mathbf{t}|_{\Sigma_S} \Rightarrow \mathbf{t} \in \pi_T\}.$$

That is, the target guarantee of a source property is that the target has the same source-level behavior, sprinkled with arbitrary target-level behavior. Conversely, the source-level obligation of a target property is the aggregate of those source traces all of whose target-level enrichments are in the target property.

Since R^S and R^T are very similar, it is simple to prove that the identity compiler $(\cdot \downarrow)$ from R^S to R^T is secure according to the trace relation \sim defined above.

Theorem 5.4 $(\cdot \downarrow \text{ is Secure } \Rightarrow \cdot \downarrow \text{ is RTC} \sim)$.

5.3 Instances of Trace-Relating Robust Preservation of Safety and Hypersafety

To provide examples of cross-language trace-relations that preserve safety and hypersafety properties, we show how existing secure compilation results can be interpreted in our framework. This indicates how the more general theory developed here can already be instantiated to encompass existing results, and that existing proof techniques can be used in order to achieve the secure compilation criteria we define.

For the preservation of safety, Patrignani and Garg [53] study a compiler from a typed, concurrent WHILE language to an untyped, concurrent WHILE language with support for memory capabilities. As in §3.3, their source has `bools` and `nats` while their target only has `nats`. Additionally, their source has an ML-like memory (where the domain is locations ℓ) while their target has an assembly-like memory (where the domain is natural numbers \mathbf{n}). Their traces consider context-program interactions and as such they are concatenations of call and return actions with parameters, which can include booleans as well as locations. Because of the aforementioned differences, they need a cross-language relation to relate source and target actions.

Besides defining a relation on traces (i.e., an instance of \sim), they also define a relation between source and target safety properties. They provide an instantiation of τ that maps all safe source traces to the related target ones. This ensures that no additional target trace is introduced in the target property, and source safety properties are mapped to target safety ones by τ . Their compiler is then proven to generate code that respects τ , so they achieve a variation of $\text{RTP}^{\text{Safe} \circ \tilde{\tau}}$.

Concerning the preservation of hypersafety, Patrignani and Garg [52] consider compilers in a reactive setting where traces are sequences of input ($\alpha?$) and output ($\alpha!$) actions. In their setting, traces are different between source and target, so they define a

cross-language relation on actions that is total on the source actions and injective. Additionally, their set of target output actions is strictly larger than the source one, as it includes a special action \checkmark , which is how compiled code must respond to invalid target inputs (i.e., receiving a **bool** when a **nat** was expected). Starting from the relation on actions, they define **TPC**, which is an instance of what we call τ . Informally, given a set of source traces, **TPC** generates all target traces that are related (pointwise) to a source trace. Additionally, it generates all traces with interleavings of undesired inputs $\alpha?$ followed by \checkmark as long as removing $\alpha?\checkmark$ leaves a trace that relates to the source trace. **TPC** preserves hypersafety across languages, i.e., it is an instance of $\text{RSCHP}^{\text{HSafe} \circ \tau}$ (mapping source hypersafety to target hypersafety (and safety to safety)).

6 Related Work

We already discussed how our results relate to some existing work in correct compilation [34, 64] and secure compilation [2, 52, 53]. We also already mentioned that most of our definitions and results make no assumptions about the structure of traces. One result that relies on the structure of traces is [Theorem 5.2](#), which involves some *finite prefix* m , suggesting traces should be some sort of sequences of events (or states), as customary when one wants to refer to safety properties [14]. It is however sufficient to fix a topology on properties where safety properties coincide with closed sets [48]. Even for reasoning about safety, hypersafety, or arbitrary hyperproperties, traces can therefore be values, sequences of program states, or of input output events, or even the recently proposed *interaction trees* [68]. In the latter case we believe that the compilation from IMP to ASM proposed by Xia et al. [68] can be seen as an instance of HC^\sim , for the relation they call “trace equivalence.”

Compilers Where Our Work Could Be Useful. Our work should be broadly applicable to understanding the guarantees provided by many verified compilers. For instance, Wang et al. [67] recently proposed a CompCert variant that compiles all the way down to machine code, and it would be interesting to see if the model at the end of [§3.1](#) applies there too. This and many other verified compilers [12, 30, 43, 61] beyond CakeML [64] deal with resource exhaustion and it would be interesting to also apply the ideas of [§3.2](#) to them. Hur and Dreyer [27] devised a correct compiler from an ML language to assembly using a cross-language logical relation to state their CC theorem. They do not have traces, though were one to add them, the logical relation on values would serve as the basis for the trace relation and therefore their result would attain CC^\sim .

Switching to more informative traces capturing the interaction between the program and the context is often used as a proof technique for secure compilation [2, 28, 51]. Most of these results consider a cross-language relation, so they probably could be proved to attain one of the criteria from [Figure 2](#).

Generalizations of Compiler Correctness. The compiler correctness definition of Morris [42] was already general enough to account for trace relations, since it considered a translation between the semantics of the source program and that of the compiled program, which he called “decode” in his diagram, reproduced in [Figure 3](#) (left). And even some of the more recent compiler correctness definitions preserve this kind of flexibility [55]. While CC^\sim can be seen as an instance of a definition by Morris [42], we are not aware of any prior work that investigated the preservation of properties when the

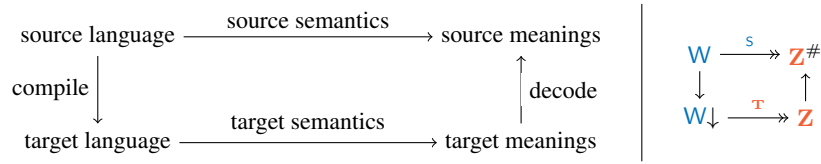


Fig. 3: Morris’s [42] (left) and Melton et al.’s [39] and Sabry and Wadler’s [59] (right)

“decode translation” is neither the identity nor a bijection, and source properties need to be re-interpreted as target ones and vice versa.

Correct Compilation and Galois Connections. Melton et al. [39] and Sabry and Wadler [59] expressed a strong variant of compiler correctness using the diagram of Figure 3 (right) [39, 59]. They require that compiled programs *parallel* the computation steps of the original source programs, which can be proven showing the existence of a *decompilation* map $\#$ that makes the diagram commute, or equivalently, the existence of an adjoint for \downarrow ($W \leq W' \iff W \rightarrow W'$ for both source and target). The “parallel” intuition can be formalized as an instance of CC^\sim . Take source and target traces to be finite or infinite sequences of program states (maximal trace semantics [15]), and relate them exactly like Melton et al. [39] and Sabry and Wadler [59]; more details can be found in §C.

Translation Validation. Translation validation is an important alternative to proving that all runs of a compiler are correct. A variant of CC^\sim for translation validation can simply be obtained by specializing the definition to a particular W , and one can obtain again the same trinitarian view. Similarly for our other criteria, including our extensions of the secure compilation criteria of Abate et al. [2], which Busi et al. [10] seem to already be considering in the context of translation validation.

7 Conclusion and Future Work

We have extended the property preservation view on compiler correctness to arbitrary trace relations, and believe that this will be useful for understanding the guarantees various compilers provide. An open question is whether, given a compiler, there exists a most precise \sim relation for which this compiler is correct. As mentioned in §1, every compiler is CC^\sim for some \sim , but under which conditions is there a most precise relation? In practice, more precision may not always be better though, as it may be at odds with compiler efficiency and may not align with more subjective notions of usefulness, leading to tradeoffs in the selection of suitable relations. Finally, another interesting direction for future work is studying whether using the relation to Galois connections allows to more easily compose trace relations for different purposes, say, for a compiler whose target language has undefined behavior, resource exhaustion, and side-channels. In particular, are there ways to obtain complex relations by combining simpler ones in a way that eases the compiler verification burden?

Acknowledgements. We thank Akram El-Korashy and Amin Timany for participating in an early discussion about this work and the anonymous reviewers for their valuable feedback. This work was in part supported by the [European Research Council](#) under [ERC Starting Grant SECOMP \(715753\)](#), by the German Federal Ministry of Education

and Research (BMBF) through funding for the CISA-Stanford Center for Cybersecurity (FKZ: 13N1S0762), by DARPA grant SSITH/HOPE (FA8650-15-C-7558) and by UAIC internal grant 07/2018.

Appendix

The appendix contains the following addenda. First of all, it presents the proof index (Appendix A). Secondly, it discusses extension of the notion of compiler correctness to other (hyper)property classes: safety properties and non-subset closed hyperproperties (Appendix B). Then, it discusses the relation with the work of Melton et al. [39] and Sabry and Wadler [59] in more details (Appendix C). Then it contains missing formalization bits from Section 3 (Appendix D). Then it contains a further discussion on the composition of relations and how that affects our criteria (Appendix E). Finally, it presents a more in-depth explanation of how to scale our results to the setting of secure compilation (Appendix F).

A Proof Index

Proof (Of Theorem 2.4 (♣)). See Corollary `Adj_σTP_iff_τTP` in `Def.v`. In general, if a program satisfies a property π , then it satisfies every extension $\pi' \supseteq \pi$. Using this, the theorem follows by:

(\Rightarrow) Assume `TP τ` and that `W` satisfies $\sigma(\pi_T)$. Apply `TP τ` to `W` and $\sigma(\pi_T)$ and deduce that `W \downarrow` satisfies $\tau(\sigma(\pi_T)) \subseteq \pi_T$.

(\Leftarrow) Assume `TP σ` and that `W` satisfies $\pi_S \subseteq \sigma(\tau(\pi_S))$. Apply `TP σ` to `W` and $\sigma(\tau(\pi_S))$ deducing `W \downarrow` satisfies $\tau(\pi_S)$.

Proof (Of Theorem 2.6 (♣)). See Theorems `rel_TC_τTP` and `rel_TC_σTP` in `TraceCriterion.v`, where the `TP τ` \iff `TP σ` part follows directly from Theorem 2.4.

Lemma A.1 (Special relations and consequences on the adjoints). *Let X, Y be two arbitrary sets and $\sim \subseteq X \times Y$. Assume \sim is a total and surjective map from Y to X . Let $\alpha \rightsquigarrow \gamma$ be its existential and universal image, i.e.*

$$\tilde{\alpha} = \lambda \pi_X. \{y \mid \exists x \in \pi_X. x \sim y\}$$

$$\tilde{\gamma} = \lambda \pi_Y. \{x \mid \forall y. x \sim y \Rightarrow y \in \pi_Y\}$$

Then $\tilde{\gamma} = \lambda \pi_Y. \{x \mid \exists y \in \pi_Y. x \sim y\}$, and $\tilde{\gamma}$ is injective.

Proof (Of Lemma A.1). See Lemma `rel_total_surjective` and `rel_total_surjective_up_inj` in `Galois.v`

Proof (Of Theorem 2.11 (♣)). `rel_TC_sClτSCHP, rel_TC_sCl_σRSCHP` in `SSCHCriterion.v`.

Proof (Of Theorem 3.3 (♣)). See Theorem `correctness` in `TypeRelationExampleInput.v`.

Proof (Of Theorem 3.4). See in §D.3.

Proof (Of Theorem 4.1). Assume $W \models ANI_\phi^\rho$ and $W \downarrow \rightsquigarrow t_1, t_2$ with $\phi^\#(t_1^\circ) = \phi^\#(t_2^\circ)$.

We have to show that $\rho^\#(t_1^\circ) = \rho^\#(t_2^\circ)$.

By CC^\sim there exists $s_1 \sim t_1$ and $s_2 \sim t_2$ such that $W \rightsquigarrow s_1, s_2$. As a preliminary, apply Lemma A.1 to the relations $\tilde{\sim} \circ \text{swap}$ and deduce g° is injective. Notice also that by functionality and totality, of $\tilde{\sim}$ and of \sim , $f^\circ(t_1^\circ) = \{s_1^\circ\}$ and $f^\circ(t_2^\circ) = \{s_2^\circ\}$ and a similar fact holds for s_2 and t_2 .

$$\begin{aligned}
\phi^\#(t_1^\circ) = \phi^\#(t_2^\circ) &\Rightarrow && \text{[definition of } \phi^\# \text{]} \\
g^\circ \circ \phi \circ f^\circ(t_1^\circ) = g^\circ \circ \phi \circ f^\circ(t_2^\circ) &\Rightarrow && \text{[} g^\circ \text{ injective]} \\
\phi \circ f^\circ(t_1^\circ) = \phi \circ f^\circ(t_2^\circ) &\Rightarrow && \text{[} f^\circ(t_i^\circ) = s_i^\circ \text{ } i = 1, 2 \text{]} \\
\phi(s_1^\circ) = \phi(s_2^\circ) &\Rightarrow && \text{[} W \models ANI_\phi^\rho \text{]} \\
\rho(s_1^\circ) = \rho(s_2^\circ) &\Rightarrow && \text{[} s_i^\circ = f^\circ(t_i^\circ) \text{ } i = 1, 2 \text{]} \\
\rho \circ f^\circ(t_1^\circ) = \rho \circ f^\circ(t_2^\circ) &\Rightarrow && \text{[functionality of } g^\circ \text{]} \\
g^\circ \circ \rho \circ f^\circ(t_1^\circ) = g^\circ \circ \rho \circ f^\circ(t_2^\circ) &\Rightarrow && \text{[definition of } \rho^\# \text{]} \\
\rho^\#(t_1^\circ) = \rho^\#(t_2^\circ), &&&
\end{aligned}$$

so that $W \downarrow \models ANI_{\phi^\#}^{\rho^\#}$.

We now show that if $\tilde{\sim}$ is surjective, i.e., g° injective, $ANI_{\phi^\#}^{\rho^\#} \subseteq Cl_{\subseteq} \circ \tilde{\tau}(ANI_\phi^\rho)$.

Let $\pi_T \in ANI_{\phi^\#}^{\rho^\#}$, we show that $\pi_T \subseteq \tilde{\tau}(\pi_S)$ for some $\pi_S \in ANI_\phi^\rho$.

The source property $\pi_S = \{s \mid \exists t \in \pi_T. s \sim t\} = f(\pi_T)$ is such that $\pi_T \subseteq \tilde{\tau}(\pi_S)$.

We only need to show $\pi_S \in ANI_\phi^\rho$. Let $s_1, s_2 \in \pi_S$,

$$\begin{aligned}
\phi(s_1^\circ) = \phi(s_2^\circ) &\Rightarrow && \text{[by } f^\circ(t^\circ) = s^\circ \text{ for some } t \in \pi_T \text{]} \\
\phi(f^\circ(t_1^\circ)) = \phi(f^\circ(t_2^\circ)) &\Rightarrow && \text{[} g^\circ \text{ is a function]} \\
g^\circ(\phi(f^\circ(t_1^\circ))) = g^\circ(\phi(f^\circ(t_2^\circ))) &\Rightarrow && \text{[by definition of } \phi^\# \text{]} \\
\phi^\#(t_1^\circ) = \phi^\#(t_2^\circ) &\Rightarrow && \text{[} \pi_T \in ANI_{\phi^\#}^{\rho^\#} \text{]} \\
\rho^\#(t_1^\circ) = \rho^\#(t_2^\circ) &\Rightarrow && \text{[definition of } \rho^\# \text{]} \\
g^\circ(\rho(f^\circ(t_1^\circ))) = g^\circ(\rho(f^\circ(t_2^\circ))) &\Rightarrow && \text{[by injectivity of } g^\circ \text{]} \\
\rho(f^\circ(t_1^\circ)) = \rho(f^\circ(t_2^\circ)) &\Rightarrow && \text{[} f^\circ(t_i) = s_i^\circ, i = 1, 2 \text{]} \\
\rho(s_1^\circ) = \rho(s_2^\circ), &&&
\end{aligned}$$

that shows $\pi_S \in ANI_\phi^\rho$ and concludes the proof.

Proof (Of Theorem 4.2). Assume $W \models ANI_\phi^\rho$ and $W \downarrow \rightsquigarrow t_1, t_2$ with $\phi^\#(t_1^\circ) = \phi^\#(t_2^\circ)$.

We have to show that $\rho^\#(t_1^\circ) = \rho^\#(t_2^\circ)$, for an arbitrary $\rho^\#$ that satisfies the condition $H \equiv \forall s t. s^\circ \sim t^\circ \Rightarrow \rho^\#(\tilde{\tau}^\circ(\rho(s^\circ))) = \rho^\#(t^\circ)$.

By CC^\sim there exists $s_1 \sim t_1$ and $s_2 \sim t_2$ such that $W \rightsquigarrow s_1, s_2$. As a preliminary, recall that Lemma A.1 ensures g° is injective. Moreover notice that by functionality and totality, of \sim , $f^\circ(t_1^\circ) = \{s_1^\circ\}$ and $f^\circ(t_2^\circ) = \{s_2^\circ\}$.

$$\begin{aligned}
\phi^\#(t_1^\circ) = \phi^\#(t_2^\circ) &\Rightarrow && \text{[definition of } \phi^\# \text{]} \\
g^\circ \circ \phi \circ f^\circ(t_1^\circ) = g^\circ \circ \phi \circ f^\circ(t_2^\circ) &\Rightarrow && \text{[} g^\circ \text{ injective]}
\end{aligned}$$

$$\begin{aligned}
\phi \circ f^\circ(\mathbf{t}_1^\circ) = \phi \circ f^\circ(\mathbf{t}_2^\circ) &\Rightarrow && [f^\circ(\mathbf{t}_i^\circ) = \mathbf{s}_i^\circ \ i = 1, 2] \\
\phi(\mathbf{s}_1^\circ) = \phi(\mathbf{s}_2^\circ) &\Rightarrow && [W \models ANI_\phi^\rho] \\
\rho(\mathbf{s}_1^\circ) = \rho(\mathbf{s}_2^\circ) &\Rightarrow && [\text{functionality of } \tilde{\tau}^\circ] \\
\tilde{\tau}^\circ(\rho(\mathbf{s}_1^\circ)) = \tilde{\tau}^\circ(\rho(\mathbf{s}_2^\circ)) &\Rightarrow && [\text{by functionality of } \rho^\#] \\
\rho^\#(\tilde{\tau}^\circ(\rho(\mathbf{s}_1^\circ))) = \rho^\#(\tilde{\tau}^\circ(\rho(\mathbf{s}_2^\circ))) &\Rightarrow && [\text{by condition } H] \\
\rho^\#(\mathbf{t}_1^\circ) = \rho^\#(\mathbf{t}_2^\circ) &&&
\end{aligned}$$

so that $W \downarrow \models ANI_{\phi^\#}^{\rho^\#}$.

Proof (Of Theorem 4.3). Assume $W \models ANI_{\phi^\#}^{\rho^\#}$ and $W \downarrow \rightsquigarrow \mathbf{t}_1, \mathbf{t}_2$ with $\phi(\mathbf{t}_1^\circ) = \phi(\mathbf{t}_2^\circ)$ and ϕ satisfying the condition $H \equiv \forall \mathbf{s} \mathbf{t}. \mathbf{s}^\circ \sim \mathbf{t}^\circ \Rightarrow \phi(\mathbf{t}^\circ) = \phi(\tilde{\tau}^\circ(\mathbf{s}^\circ))$. We have to show that $\rho(\mathbf{t}_1^\circ) = \rho(\mathbf{t}_2^\circ)$. By CC^\sim there exists $\mathbf{s}_1 \sim \mathbf{t}_1$ and $\mathbf{s}_2 \sim \mathbf{t}_2$ such that $W \rightsquigarrow \mathbf{s}_1, \mathbf{s}_2$. As a preliminary, recall that Lemma A.1 ensures $\tilde{\sigma}^\circ$ is injective. Moreover notice that by functionality and totality, of \sim , $\tilde{\tau}^\circ(\mathbf{s}_1^\circ) = \{\mathbf{t}_1^\circ\}$ and $\tilde{\tau}^\circ(\mathbf{s}_2^\circ) = \{\mathbf{t}_2^\circ\}$.

$$\begin{aligned}
\phi(\mathbf{t}_1^\circ) = \phi(\mathbf{t}_2^\circ) &\Rightarrow && [\text{by } H] \\
\phi(\tilde{\tau}^\circ(\mathbf{s}_1^\circ)) = \phi(\tilde{\tau}^\circ(\mathbf{s}_2^\circ)) &\Rightarrow && [\text{functionality of } \tilde{\sigma}^\circ] \\
\tilde{\sigma}^\circ(\phi(\tilde{\tau}^\circ(\mathbf{s}_1^\circ))) = \tilde{\sigma}^\circ(\phi(\tilde{\tau}^\circ(\mathbf{s}_2^\circ))) &\Rightarrow && [\text{definition of } \phi^\#] \\
\phi^\#(\mathbf{s}_1^\circ) = \phi^\#(\mathbf{s}_2^\circ) &\Rightarrow && [W \models ANI_{\phi^\#}^{\phi^\#}] \\
\rho^\#(\mathbf{s}_1^\circ) = \rho^\#(\mathbf{s}_2^\circ) &\Rightarrow && [\text{by definition of } \rho^\#] \\
\tilde{\sigma}^\circ(\rho^\#(\tilde{\tau}^\circ(\mathbf{s}_1^\circ))) = \tilde{\sigma}^\circ(\rho^\#(\tilde{\tau}^\circ(\mathbf{s}_2^\circ))) &\Rightarrow && [\text{injectivity of } \tilde{\sigma}^\circ] \\
\rho(\tilde{\tau}^\circ(\mathbf{s}_1^\circ)) = \rho(\tilde{\tau}^\circ(\mathbf{s}_2^\circ)) &\Rightarrow && [\tilde{\tau}^\circ(\mathbf{s}_i^\circ) = \{\mathbf{t}_i^\circ\} \ i = 1, 2] \\
\rho(\mathbf{t}_1^\circ) = \rho(\mathbf{t}_2^\circ) &&&
\end{aligned}$$

so that $W \downarrow \models ANI_\phi^\rho$.

Proof (Of Theorem 5.1 (♣)). Theorems `rel_RTC_τRTP` and `rel_RTC_σRTP` in `RobustTraceCriterion.v`.

Proof (Of Theorem 5.2 (♣)). Theorems `tilde_RSC_σRSP` and `tilde_RSC_CI_τRTP` in `RobustSafetyCriterion.v`.

Proof (Of Theorem 5.3 (♣)). Lemmas `σRHP_rel_RHC` and `rel_RHC_σRHP` and Theorem `rel_RHC_τRHP` in `RobustHyperCriterion.v`.

Proof (Of Theorem 5.4 (♣)). (See theorem `extra_target_RTCT` in `MoreTargetEventsExample.v`, mechanizing a slightly simplified model.) By definition of RTC^\sim we need to find a source context and source trace given a source program, target context and target trace related by compilation and program semantics: This instantiation is simple since the trace relation is a *function* from target traces to source traces, and it is easy to clean target contexts to produce equivalent source context without target-only events. The proof is a trivial instance of *precise, context-based backtranslation* [2, 47, 54, 62], aided by a few straightforward lemmas and where the case of function calls is guaranteed to terminate by the language.

Proof (Of Theorem B.3 (2)). Theorems `tilde_SC_σSP` and `tilde_SC_CI_τTP` in `SafetyCriterion.v`.

Proof (Of Theorem B.4). For the implication from left to right, assume $W \models H$. By $CC^=$ have $\mathbf{beh}(W\downarrow) = \mathbf{beh}(W)$, so that $W\downarrow \models H$ as well.

For the implication from right to left, instantiate HP with the hyperproperty $\{\mathbf{beh}(W)\}$, for a given W , and deduce that $W\downarrow \models \{\mathbf{beh}(W)\}$ i.e., $\mathbf{beh}(W\downarrow) = \mathbf{beh}(W)$.

Proof (Of Theorem B.5 (2)). Theorems `rel_HC_τHP`, `rel_HC_σHP` and `σHP_rel_HC` in `HyperCriterion.v`.

B Preserving Other (Hyper)Property Classes

In this section we investigate how to preserve two classes of (hyper)properties beyond trace properties: safety properties (§B.1) and arbitrary hyperproperties that are not just subset-closed (§B.2). For each of these classes, we start by giving an intuition of what it means to preserve such a class in the equal-trace setting, then we study preservation of that class in the trace-relating setting. In both cases, we obtain a somewhat weaker preservation result that reflects the loss of precision incurred by the properties.

B.1 Preserving Safety Properties

Safety collects all trace properties prescribing that “something bad never happens,” so that their violation can be monitored and, when observed, no longer restored [14]. In order to define this class, we assume traces are built in a standard manner as lists or streams of events, so that we can consider *finite prefixes* m of a trace t , written $m \leq t$.

Definition B.1 (Safety Properties). A property π is a safety property if $\forall t \notin \pi. \exists m \leq t. \forall t'. m \leq t' \Rightarrow t' \notin \pi$.

A proof of the preservation of all trace properties, i.e., of $CC^=$, may require to show that a source program can produce an infinite stream of events. In order to prove the same for safety properties, one only has to show the following weaker variant of $CC^=$:

$$SC^= \equiv \forall W, m. W\downarrow \rightsquigarrow m \Rightarrow W \rightsquigarrow m$$

where $W \rightsquigarrow m$ means that W produces the finite prefix m , i.e., that m is a prefix of a trace produced by W , that is $\exists t. m \leq t \wedge W \rightsquigarrow t$.

Recalling the intention of safety properties to prescribe “something bad never happens,” $W \rightsquigarrow m$ can be interpreted as a violation of a safety property, with m being a bad prefix. The statement of $SC^=$ can therefore be read as “whenever $W\downarrow$ violates a safety property, then W does.” By contraposition, this was proven equivalent to the preservation of arbitrary safety properties [2]:

$$SC^= \iff \forall \pi \in \mathit{Safety}. W \models \pi \Rightarrow W\downarrow \models \pi$$

In the trace-relating setting, we generalize $SC^=$ by requiring that if $W\downarrow$ violates a target safety property $\pi_{\mathbf{T}} \in \mathit{Safety}_{\mathbf{T}}$, then W violates its source interpretation, $\tilde{\sigma}(\pi_{\mathbf{T}})$.

Definition B.2 (SC^{\sim}). Given a trace relation $\sim \subseteq \mathit{Traces}_{\mathbf{S}} \times \mathit{Trace}_{\mathbf{T}}$, a compilation chain \downarrow is SC^{\sim} if and only if the following holds:

$$SC^{\sim} \equiv \forall W. \forall m. W\downarrow \rightsquigarrow m \Rightarrow \exists t. \exists s. m \leq t \wedge s \sim t \wedge W \rightsquigarrow s$$

Again, we propose two generalizations of the preservation of safety properties that are equivalent to SC^\sim . Similarly to what we had to do in §2.3, we have to take into account the fact that for $\pi_S \in \mathbf{Safety}_S$, $\tilde{\tau}(\pi_S)$ is not necessarily in \mathbf{Safety}_T . Hence, we have to close $\tilde{\tau}(\pi_S)$ in the class of target safety properties.

Theorem B.3 (Trinitarian view for Safety). *For a trace relation $\sim \subseteq \mathbf{Traces}_S \times \mathbf{Trace}_T$ and its induced property mappings $\tilde{\sigma}$ and $\tilde{\tau}$, SC^\sim is equivalent to these criteria:*

$$\begin{array}{l} SP^{\tilde{\sigma}} \equiv \forall W \forall \pi_T \in \mathbf{Safety}_T. W \models \tilde{\sigma}(\pi_T) \Rightarrow W \downarrow \models \pi_T \\ TP^{Safe \circ \tilde{\tau}} \equiv \forall W \forall \pi_S \in 2^{\mathbf{Traces}_S}. W \models (\pi_S) \Rightarrow W \downarrow \models (Safe \circ \tilde{\tau})(\pi_S) \end{array} \quad \begin{array}{c} SC^\sim \\ \swarrow \quad \searrow \\ SP^{\tilde{\sigma}} \iff TP^{Safe \circ \tilde{\tau}} \end{array}$$

where $Safe(\pi_T) = \bigcap \{S_T \mid S_T \in \mathbf{Safety}_T \wedge \pi_T \subseteq S_T\}$ is the smallest target safety property that contains π_T .

Notice that in $TP^{Safe \circ \tilde{\tau}}$, we quantify over arbitrary source properties, but the guarantee we get in the target is given by approximating the guarantee defined by $\tilde{\tau}$, with a safety property. This second approximation is optimal by minimality of $Safe$, and is necessary as $\tilde{\tau}$ may not map safety properties to safety properties (in this case, no comparison with SC^\sim would be possible).

B.2 Preserving Non-Subset Closed Hyperproperties

Many hyperproperties of interest, including possibilistic information-flow policy are not subset-closed [14]. CC^\equiv is not enough to guarantee the preservation of such hyperproperties, as it only ensures refinement of the behaviors through compilation but not the other inclusion, sometimes referred to as *reflection*. It is easy to show that that the following strengthening of CC^\equiv is equivalent to the preservation of arbitrary hyperproperties.

Theorem B.4 (HC^\equiv , HP). *For every compilation chain \downarrow , the followings are equivalent*

$$HC^\equiv \equiv \forall W. \mathbf{beh}(W \downarrow) = \mathbf{beh}(W) \quad HP \equiv \forall W \forall H \in 2^{2^{\mathbf{Trace}}}. W \models H \Rightarrow W \downarrow \models H$$

The generalization to the trace-relating setting, $HC^\sim \equiv \forall W. \mathbf{beh}(W \downarrow) = \tilde{\tau}(\mathbf{beh}(W))$, does not come with two equivalent formulations in terms of preservation of hyperproperties. While for $HP^{\tilde{\tau}}$ the equivalence holds unconditionally, for $HP^{\tilde{\sigma}}$ some extra assumptions are required, depending on the implication one is interested in. It is still possible, and correct, to deduce a source obligation for a given target hyperproperty H_T when no information is lost in the the composition $\tilde{\tau} \circ \tilde{\sigma}$ (i.e., the two maps are a Galois *insertion*). *Vice versa*, HC^\sim is a consequence of $HP^{\tilde{\sigma}}$ when no information is lost in composing in the other direction, $\tilde{\sigma} \circ \tilde{\tau}$ (i.e., the two maps are a Galois *reflection*).

Theorem B.5 (Weak Trinity). *Given a trace relation $\sim \subseteq \mathbf{Traces}_S \times \mathbf{Trace}_T$, and its induced property mappings $\tilde{\sigma}$ and $\tilde{\tau}$, HC^\sim is equivalent to $HP^{\tilde{\tau}}$ below. Moreover if $\tilde{\tau} \hookrightarrow \tilde{\sigma}$ is a Galois insertion (i.e., $\tilde{\tau} \circ \tilde{\sigma} = id$), HC^\sim implies $HP^{\tilde{\sigma}}$, while if $\tilde{\sigma} \hookrightarrow \tilde{\tau}$ is a Galois reflection (i.e., $\tilde{\sigma} \circ \tilde{\tau} = id$), $HP^{\tilde{\sigma}}$ implies HC^\sim .*

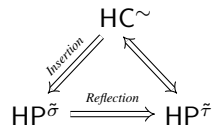
$$\begin{aligned}
\text{HP}^{\tilde{\tau}} &\equiv \forall W \forall H_S. W \models H_S \Rightarrow W \downarrow \models \tilde{\tau}(H_S) \\
\text{HP}^{\tilde{\sigma}} &\equiv \forall W \forall H_T. W \models \tilde{\sigma}(H_T) \Rightarrow W \downarrow \models H_T
\end{aligned}$$


Figure 4 sums up our generalizations and orders them according to their strength.

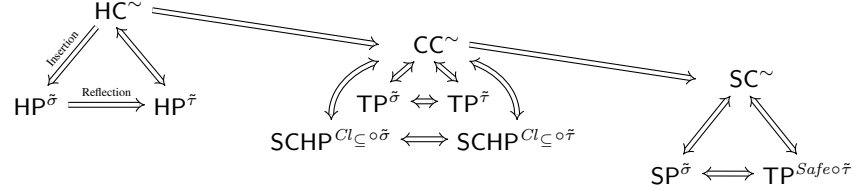


Fig. 4: Generalization of Compiler Correctness and its trace-relating variations.

C Relation to Melton et al. and Sabry and Wadler

In this section we assume that source and target languages are equipped with operational small steps semantics, \rightarrow and \rightarrow respectively. Let States_S and State_T be the set of all possible program states in the source and in the target respectively. We define traces as finite or infinite sequences of program states, formally

$$\text{Traces}_S = \text{States}_S^* \cup \text{States}_S^\omega$$

$$\text{Trace}_T = \text{State}_T^* \cup \text{State}_T^\omega.$$

For both source and target languages, we say that a program produces a trace if and only if it can be extracted by an admissible execution of the program itself. Formally

$$\begin{aligned}
W \rightsquigarrow_S &\iff \exists (W_0 \rightarrow W_1 \rightarrow \dots \rightarrow W_k \rightarrow \dots). \\
&\quad \exists (0 < i_1 < i_2 < \dots < i_k < \dots). \\
&\quad W_0 = s_0 \wedge \forall j. s_j = W_{i_j}.
\end{aligned}$$

a similar definition is given for the target, based on the target operational semantics \rightarrow . The compilation induces a relation between source and target states,

$$s_k \sim t_h \iff s_k \downarrow = t_h$$

that we lift to traces by requiring the traces to be pointwise related,

$$s \sim t \iff \forall n. s_n \downarrow = t_n$$

Recall the definition of CC^\sim

$$\text{CC}^\sim \equiv \forall W t. W \downarrow \rightsquigarrow t \Rightarrow \exists s \sim t. W \rightsquigarrow_S$$

that spelled out states that anytime we extract some states from the execution of $W \downarrow$, it is possible to extract the same number of source, and related, states from some execution of W . In other words CC^\sim requires that whenever $W \downarrow$ reduces in an arbitrary (but finite) amount of steps to some target state, $W \downarrow \rightarrow^* t_k$, then $W \rightarrow^* s_h$ for some state $s_h \sim t_k$, i.e., $s_h \downarrow = t_k$.

This captures in general a weaker requirement than the existence of the decompilation function in Melton et al. [39] and Sabry and Wadler [59], but coincide for the case study in Melton et al. [39], where \downarrow is an injective map.

D Elided Formalization Bits

D.1 Additional Formalism for Section 3.3

The Source Language Below is the syntax of the source language.

$$e ::= n \mid b \mid e \text{ op } e \mid \text{if } e \text{ then } e \text{ else } e \mid \text{in}_b \mid \text{in}_n \quad \text{op} ::= + \mid \times \mid \leq \quad \text{ty} ::= B \mid N$$

$$r ::= n \mid b \mid \text{error} \quad i ::= n \mid b \quad \text{is} ::= i \cdot \text{is} \mid \emptyset \quad s ::= \langle \text{is}, r \rangle$$

Below are the typing rules for the source language.

$$\begin{array}{c}
\begin{array}{c} \text{(Type-nat)} \\ \hline \vdash n : N \end{array} \quad \begin{array}{c} \text{(Type-bool)} \\ \hline \vdash b : B \end{array} \quad \begin{array}{c} \text{(Type-plus)} \\ \hline \vdash e_1 : N \quad \vdash e_2 : N \\ \hline \vdash e_1 + e_2 : N \end{array} \quad \begin{array}{c} \text{(Type-times)} \\ \hline \vdash e_1 : N \quad \vdash e_2 : N \\ \hline \vdash e_1 \times e_2 : N \end{array} \\
\begin{array}{c} \text{(Type-ite)} \\ \hline \vdash e_1 : B \quad \vdash e_2 : \text{ty} \quad \vdash e_3 : \text{ty} \\ \hline \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \text{ty} \end{array} \quad \begin{array}{c} \text{(Type-le)} \\ \hline \vdash e_1 : N \quad \vdash e_2 : N \\ \hline \vdash e_1 \leq e_2 : B \end{array} \quad \begin{array}{c} \text{(Type-in-b)} \\ \hline \vdash \text{in}_b : B \end{array} \\
\begin{array}{c} \text{(Type-in-n)} \\ \hline \vdash \text{in}_n : N \end{array}
\end{array}$$

Well-typed programs do not produce *error* (error), see Theorem `type_soundness` in file `TypeRelationExampleInput.v`.

Below is the big-step semantics of the source language.

$$\begin{array}{c}
\begin{array}{c} \text{(Sem-nat)} \\ \hline n \rightsquigarrow \langle \emptyset, n \rangle \end{array} \quad \begin{array}{c} \text{(Sem-in-nat)} \\ \hline \text{in}_n \rightsquigarrow \langle n \cdot \emptyset, n \rangle \end{array} \quad \begin{array}{c} \text{(Sem-bool)} \\ \hline b \rightsquigarrow \langle \emptyset, b \rangle \end{array} \quad \begin{array}{c} \text{(Sem-in-bool)} \\ \hline \text{in}_b \rightsquigarrow \langle b \cdot \emptyset, b \rangle \end{array} \\
\begin{array}{c} \text{(Sem-op-nat)} \\ \hline e_1 \rightsquigarrow \langle \text{is}_1, n_1 \rangle \quad e_2 \rightsquigarrow \langle \text{is}_2, n_2 \rangle \quad \text{op} \in \{+, \times\} \\ \hline e_1 \text{ op } e_2 \rightsquigarrow \langle \text{is}_1 \cdot \text{is}_2, (n_1 \text{ op } n_2) \rangle \end{array} \\
\begin{array}{c} \text{(Sem-le)} \\ \hline e_1 \rightsquigarrow \langle \text{is}_1, n_1 \rangle \quad e_2 \rightsquigarrow \langle \text{is}_2, n_2 \rangle \\ \hline e_1 \leq e_2 \rightsquigarrow \langle \text{is}_1 \cdot \text{is}_2, (n_1 \leq n_2) \rangle \end{array} \quad \begin{array}{c} \text{(Sem-ite-true)} \\ \hline e_1 \rightsquigarrow \langle \text{is}_1, \text{true} \rangle \quad e_2 \rightsquigarrow \langle \text{is}_2, r_2 \rangle \\ \hline \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \rightsquigarrow \langle \text{is}_1 \cdot \text{is}_2, r_2 \rangle \end{array} \\
\begin{array}{c} \text{(Sem-ite-false)} \\ \hline e_1 \rightsquigarrow \langle \text{is}_1, \text{false} \rangle \quad e_3 \rightsquigarrow \langle \text{is}_3, r_3 \rangle \\ \hline \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \rightsquigarrow \langle \text{is}_1 \cdot \text{is}_3, r_3 \rangle \end{array}
\end{array}$$

The Target Language Below is the syntax of the target language.

$$e ::= n \mid e \text{ op } e \mid \text{if } e \leq e \text{ then } e \text{ else } e \mid \text{in}_n \quad \text{op} ::= + \mid \times$$

$$r ::= n \quad i ::= n \quad \text{is} ::= i \cdot \text{is} \mid \emptyset \quad t ::= \langle \text{is}, r \rangle$$

Below is the big-step semantics of the target language. Rules `Sem-nat`, `Sem-in-nat` and `Sem-op-nat` are the same as for the source and we present the new rules only:

$$\begin{array}{c}
\begin{array}{c} \text{(Sem-itele-true)} \\ \hline e_1 \rightsquigarrow \langle \text{is}_1, n_1 \rangle \quad e_2 \rightsquigarrow \langle \text{is}_2, n_2 \rangle \\ n_1 \leq n_2 \quad e_3 \rightsquigarrow \langle \text{is}_3, n_3 \rangle \\ \hline \text{if } e_1 \leq e_2 \text{ then } e_3 \text{ else } e_4 \rightsquigarrow \langle \text{is}_1 \cdot \text{is}_2 \cdot \text{is}_3, n_3 \rangle \end{array} \\
\begin{array}{c} \text{(Sem-ite-false)} \\ \hline e_1 \rightsquigarrow \langle \text{is}_1, n_1 \rangle \quad e_2 \rightsquigarrow \langle \text{is}_2, n_2 \rangle \\ n_1 > n_2 \quad e_4 \rightsquigarrow \langle \text{is}_4, n_4 \rangle \\ \hline \text{if } e_1 \leq e_2 \text{ then } e_3 \text{ else } e_4 \rightsquigarrow \langle \text{is}_1 \cdot \text{is}_2 \cdot \text{is}_4, n_4 \rangle \end{array}
\end{array}$$

The Compiler

$$\begin{aligned}
n \downarrow &= \mathbf{n} & \text{true} \downarrow &= \mathbf{1} & e_1 + e_2 \downarrow &= e_1 \downarrow + e_2 \downarrow \\
\text{in}_n \downarrow &= \mathbf{in}_n & \text{false} \downarrow &= \mathbf{0} & e_1 \leq e_2 \downarrow &= \text{if } e_1 \downarrow \leq e_2 \downarrow \text{ then } \mathbf{1} \text{ else } \mathbf{0} \\
\text{in}_b \downarrow &= \mathbf{in}_n & e_1 \times e_2 \downarrow &= e_1 \downarrow \times e_2 \downarrow & \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \downarrow &= \text{if } e_1 \downarrow \leq \mathbf{0} \text{ then } e_3 \downarrow \text{ else } e_2 \downarrow
\end{aligned}$$

D.2 Additional Formalism for Section 3.4

The Source Language \mathbb{N}^P \mathbb{N}^P is a statically-typed language with expressions, commands and simple types (natural numbers and pairs) whose syntax is presented below. The key aspect of \mathbb{N}^P is the primitive for sending *pairs* over a network interface. The type system of \mathbb{N}^P is unsurprising.

$$\begin{aligned}
c ::= & \text{skip} \mid \text{ifz } e \text{ then } c \text{ else } c \mid c; c \mid \text{send } e & v ::= & n \mid \langle v, v \rangle & \tau ::= & \mathbb{N} \mid \tau \times \tau \\
e ::= & n \mid e \oplus e \mid \langle e, e \rangle \mid e.1 \mid e.2 & \mathbb{E} ::= & [\cdot] \mid \mathbb{E} \oplus e \mid n \oplus \mathbb{E} \mid \mathbb{E}.1 \mid \mathbb{E}.2 \mid \langle \mathbb{E}, e \rangle \mid \langle v, \mathbb{E} \rangle
\end{aligned}$$

$$\begin{array}{c}
\begin{array}{c} \text{(Type-}\mathbb{N}^P\text{-skip)} \\ \frac{}{\vdash \text{skip}} \end{array} \quad \begin{array}{c} \text{(Type-}\mathbb{N}^P\text{-seq)} \\ \frac{\vdash c \quad \vdash c'}{\vdash c; c'} \end{array} \quad \begin{array}{c} \text{(Type-}\mathbb{N}^P\text{-send)} \\ \frac{\vdash e : \tau \times \tau'}{\vdash \text{send } e} \end{array} \quad \begin{array}{c} \text{(Type-}\mathbb{N}^P\text{-if)} \\ \frac{\vdash e : \mathbb{N} \quad \vdash c \quad \vdash c'}{\vdash \text{ifz } e \text{ then } c \text{ else } c'} \end{array} \\
\text{(Type-}\mathbb{N}^P\text{-n)} \quad \text{(Type-}\mathbb{N}^P\text{-op)} \quad \text{(Type-}\mathbb{N}^P\text{-pair)} \quad \text{(Type-}\mathbb{N}^P\text{-p1)} \\
\frac{}{\vdash n : \mathbb{N}} \quad \frac{\vdash e : \mathbb{N} \quad \vdash e' : \mathbb{N}}{\Gamma \vdash e \oplus e' : \mathbb{N}} \quad \frac{\vdash e : \tau \quad \vdash e' : \tau'}{\Gamma \vdash \langle e, e' \rangle : \tau \times \tau'} \quad \frac{}{\vdash e.1 : \tau} \\
\text{(Type-}\mathbb{N}^P\text{-p2)} \\
\frac{\vdash e : \tau \times \tau'}{\vdash e.2 : \tau'}
\end{array}$$

\mathbb{N}^P has a contextual, small-step, call by value semantics for expressions ($e \hookrightarrow e'$) and a big-step semantics for commands ($c \xrightarrow{s} c'$) that produces traces s , i.e., sequences of messages \mathbb{M} (i.e., pairs $\langle v, v \rangle$) sent over the network. In the following, queuing the empty element is ineffective: $s = \epsilon \cdot s$.

$$\begin{array}{c}
\begin{array}{c} \text{(Eval-}\mathbb{N}^P\text{-Op)} \\ \frac{n'' = n \oplus n'}{n \oplus n' \hookrightarrow n''} \end{array} \quad \begin{array}{c} \text{(Eval-}\mathbb{N}^P\text{-P1)} \\ \frac{}{\langle v, v' \rangle.1 \hookrightarrow v} \end{array} \quad \begin{array}{c} \text{(Eval-}\mathbb{N}^P\text{-P2)} \\ \frac{}{\langle v, v' \rangle.2 \hookrightarrow v'} \end{array} \quad \begin{array}{c} \text{(Eval-}\mathbb{N}^P\text{-Ctx)} \\ \frac{e \hookrightarrow e'}{\mathbb{E}[e] \hookrightarrow \mathbb{E}[e']} \end{array} \\
\text{(E-}\mathbb{N}^P\text{-skip)} \quad \text{(E-}\mathbb{N}^P\text{-seq)} \\
\frac{}{\text{skip} \xrightarrow{\epsilon} \text{skip}} \quad \frac{c \xrightarrow{s} \text{skip} \quad c' \xrightarrow{s'} \text{skip}}{c; c' \xrightarrow{s \cdot s'} \text{skip}} \\
\text{(Eval-}\mathbb{N}^P\text{-If-t)} \quad \text{(Eval-}\mathbb{N}^P\text{-If-f)} \\
\frac{e \hookrightarrow^* \mathbf{0} \quad c \xrightarrow{s} \text{skip}}{\text{if } e \text{ then } c \text{ else } c' \xrightarrow{s} \text{skip}} \quad \frac{e \hookrightarrow^* n \neq \mathbf{0} \quad c' \xrightarrow{s} \text{skip}}{\text{if } e \text{ then } c \text{ else } c' \xrightarrow{s} \text{skip}} \\
\text{(Eval-}\mathbb{N}^P\text{-Send)} \quad \text{(Sem-}\mathbb{N}^P\text{)} \\
\frac{e \hookrightarrow^* \langle v, v' \rangle}{\text{send } e \xrightarrow{\langle v, v' \rangle} \text{skip}} \quad \frac{c \xrightarrow{s} \text{skip}}{c \rightsquigarrow s}
\end{array}$$

The Target Language \mathbb{N}^n \mathbb{N}^n is a statically-typed language with the same types as \mathbb{N}^P , but its primitive for sending over a network interface only sends *natural numbers*. Nothing changes regarding statements, terms, values, types and evaluation contexts between \mathbb{N}^n and \mathbb{N}^P . There is only one typing rule that changes compared to \mathbb{N}^P , the one

for **send n**. \mathbf{N}^n has the same dynamic semantics as \mathbf{N}^p , the only changes regard the nature of messages and the reduction of **send n**.

$$\mathbf{M} ::= \mathbf{n} \quad \frac{\text{(Type-}\mathbf{N}^n\text{-send)}}{\frac{}{\vdash e : \mathbf{N}}} \quad \frac{\text{(Eval-}\mathbf{N}^n\text{-Send)}}{\frac{}{\text{send } n \xrightarrow{\mathbf{n}} \text{skip}}}$$

The Compiler from \mathbf{N}^p to \mathbf{N}^n : $(\cdot) \downarrow$ The compiler from \mathbf{N}^p to \mathbf{N}^n is defined inductively on the type derivation of a source statement $(\cdot \downarrow : \vdash c \rightarrow c)$ which, in turn, relies on compilation of expressions $(\cdot \downarrow : \vdash e : \tau \rightarrow e)$. The only interesting case is when compiling a **send e**, where we use the source type information concerning the message (i.e., a pair) being sent to deconstruct that pair into a sequence of natural numbers, which is what is sent in the target.

The compiler operates on *type derivations* for terms. Thus, compiling $c; c'$ would look like the following (using \mathbf{D} as a metavariable to range over derivations).

$$\left(\frac{\frac{\mathbf{D}}{\vdash c} \quad \frac{\mathbf{D}'}{\vdash c'}}{\vdash c; c'} \right) \downarrow = \left(\frac{\mathbf{D}}{\vdash c} \right) \downarrow ; \left(\frac{\mathbf{D}'}{\vdash c'} \right) \downarrow$$

However, note that each judgment uniquely identifies which typing rule is being applied and the underlying derivation. Thus, for compactness, we only write the judgment in the compilation and implicitly apply the related typing rule to obtain the underlying judgments for recursive calls.

$$\begin{aligned} (\vdash n : \mathbf{N}) \downarrow &= \mathbf{n} & (\vdash e.1 : \tau) \downarrow &= (\vdash e : \tau \times \tau') \downarrow .1 \\ (\vdash e \oplus e' : \mathbf{N}) \downarrow &= (\vdash e : \mathbf{N}) \downarrow \oplus (\vdash e' : \mathbf{N}) \downarrow & (\vdash e.2 : \tau') \downarrow &= (\vdash e : \tau \times \tau') \downarrow .2 \\ (\vdash \langle e, e' \rangle : \tau \times \tau') \downarrow &= \langle (\vdash e : \tau) \downarrow, (\vdash e' : \tau') \downarrow \rangle \\ \left(\frac{}{\vdash \text{ifz } e} \right) \downarrow &= \text{ifz } (\vdash e : \mathbf{N}) \downarrow & (\vdash \text{skip}) \downarrow &= \text{skip} \\ \left(\frac{}{\vdash \text{then } c \text{ else } c'} \right) \downarrow &= \text{then } (\vdash c) \downarrow \text{ else } (\vdash c') \downarrow \\ (\vdash \text{send } e) \downarrow &= \text{gensend } ((\vdash e : \tau \times \tau') \downarrow) & (\vdash c; c') \downarrow &= (\vdash c) \downarrow ; (\vdash c') \downarrow \end{aligned}$$

The $\text{gensend}(\cdot)$ function takes a source expression and returns a sequence of instructions sending each element of the expression. Formally: $\text{gensend}(\cdot) : \vdash e : \tau \rightarrow \mathbf{c}$

$$\text{gensend}(\vdash e : \tau) = \begin{cases} \text{send } (\vdash e : \mathbf{N}) \downarrow & \text{if } \tau = \mathbf{N} \\ \text{gensend}(\vdash e.1 : \tau'); \text{gensend}(\vdash e.2 : \tau'') & \text{if } \tau = \tau' \times \tau'' \end{cases}$$

D.3 Additional Lemmas and Proofs of Section 3.4

To prove that the compiler is correct (Theorem 3.4) we need two auxiliary results. Lemma D.1 tells us that the way we break down a source send into multiple target ones is correct. Lemma D.2 tells us that the compilation of expressions is correct.

Lemma D.1 (*gensend*(\cdot) works). *if $\text{gensend}((\vdash e : \tau \times \tau') \downarrow) \xrightarrow{\mathbf{t}} \text{skip}$ then $e \hookrightarrow^* \mathbf{v}$ and $\mathbf{v} \sim \mathbf{t}$*

Proof. We proceed by induction on τ and τ' :

$\tau = \mathbf{N}$ and $\tau' = \mathbf{N}$ By canonicity we have that $e = \langle n, n' \rangle$
 $\text{gensend}(\cdot)$ translates that into **send** $(\vdash \langle n, n' \rangle .1 : \mathbf{N}) \downarrow ; \text{send } (\vdash \langle n, n' \rangle .2 : \mathbf{N}) \downarrow$.
 By the target semantics and Rule **Eval- \mathbf{N}^n -Send**, that produces $\mathbf{t} = \mathbf{n}; \mathbf{n}'$.
 We need to prove that

– $e \equiv \langle n, n' \rangle \hookrightarrow^* \langle n, n; \rangle$ which is trivially true;
 – $\langle n, n' \rangle \sim \mathbf{n}; \mathbf{n}'$, which holds by Rule **Trace-Rel-N-N**.
 $\tau = \tau_1 \times \tau'_1$ and $\tau' = \tau_2 \times \tau'_2$ So by canonicity $e = \langle \langle e_1, e'_1 \rangle, \langle e_2, e'_2 \rangle \rangle$.
 By definition of $\text{gensend}(\cdot)$:
 $\text{gensend}(\langle \langle e_1, e'_1 \rangle, \langle e_2, e'_2 \rangle \rangle : \tau \times \tau') \downarrow$
 $= \text{gensend}(\langle \langle e_1, e'_1 \rangle, \langle e_2, e'_2 \rangle .1 : \tau \rangle \downarrow); \text{gensend}(\langle \langle e_1, e'_1 \rangle, \langle e_2, e'_2 \rangle .2 : \tau' \rangle \downarrow)$
 By assumption we have HP1 $\text{gensend}(\langle \langle e_1, e'_1 \rangle, \langle e_2, e'_2 \rangle .1 : \tau \rangle \downarrow); \xrightarrow{t_1; t_2} \text{skip}$
 $\text{gensend}(\langle \langle e_1, e'_1 \rangle, \langle e_2, e'_2 \rangle .2 : \tau' \rangle \downarrow)$
 We apply the induction hypothesis and we get
IH-SV1 $\langle \langle e_1, e'_1 \rangle, \langle e_2, e'_2 \rangle \rangle .1 \hookrightarrow^* \langle v_1, v'_1 \rangle$
IH-SV2 $\langle \langle e_1, e'_1 \rangle, \langle e_2, e'_2 \rangle \rangle .2 \hookrightarrow^* \langle v_2, v'_2 \rangle$
IH-SR1 $\langle v_1, v'_1 \rangle \sim t_1$
IH-SR2 $\langle v_2, v'_2 \rangle \sim t_2$
 We need to prove
 – $\langle \langle e_1, e'_1 \rangle, \langle e_2, e'_2 \rangle \rangle \hookrightarrow^* \langle \langle v_1, v'_1 \rangle, \langle v_2, v'_2 \rangle \rangle$
 By IH-SV1, IH-SV2 and determinism of the semantics.
 – $\langle \langle v_1, v'_1 \rangle, \langle v_2, v'_2 \rangle \rangle \sim t_1; t_2$
 This holds by Rule **Trace-Rel-M-M** with IH-SR1 and IH-SR2, for $M = \langle v_1, v'_1 \rangle$
 and $M' = \langle v_2, v'_2 \rangle$.
 $\tau = \mathbf{N}$ and $\tau' = \tau_1 \times \tau_2$ Analogous to the other cases, by IH and Rule **Trace-Rel-N-M**.
 $\tau = \tau_1 \times \tau_2$ and $\tau' = \mathbf{N}$ Analogous to other cases, by IH and Rule **Trace-Rel-M-N**.
□

Lemma D.2 ($(\cdot) \downarrow$ is correct for expressions). *if $\langle \vdash e : \tau \rangle \downarrow \hookrightarrow^* \langle \vdash v : \tau \rangle \downarrow$ then $e \hookrightarrow^* v$*

Proof. Trivial induction on the typing derivation of e .

Proof (of Theorem 3.4). By definition we need to prove that

$$\forall c, t. \text{ if } \langle \vdash c \rangle \downarrow \rightsquigarrow t \text{ then } \exists s \sim t \text{ and } c \rightsquigarrow Q$$

We proceed by induction on the typing derivation of c .

Base. $c = \text{skip}$ Trivial, as no trace is emitted.

Inductive. $c = c; c'$ By IH.

$c = \text{if } e \text{ then } c \text{ else } c'$ By Lemma D.2 and IH.

$c = \text{send } e$ By HP we have that HPQ $\langle \vdash \text{send } e \rangle \downarrow \xrightarrow{t} \text{skip}$.

We can then apply Lemma D.1 with HPQ and we have HPV $v \sim t$ and HPS $e \hookrightarrow^* v$.

By definition of $(\cdot) \downarrow$ and of \rightsquigarrow we need to prove that $\text{send } t \xrightarrow{v} \text{skip}$ and $v \sim t$.

The former holds by HPS and Rule **Eval-NP-Send**.

The latter holds by HPV. □

D.4 Additional Formalism for Section 5.2

Syntax of \mathbf{R}^S and \mathbf{R}^T

$e ::= \dots \mid e_1; e_2 \mid f(e) \mid \text{outs } n \mid \text{arg} \quad fs ::= \langle f_1, e_1 \rangle, \dots, \langle f_n, e_n \rangle \quad e ::= \dots \mid \text{out}_T n$
 $P ::= \langle fs, e \rangle \quad C ::= fs \quad W ::= C[P] \quad i ::= \dots \mid \text{outs } n \quad i ::= \dots \mid \text{out}_T n$

E Composition results

In this appendix we deduce some insights from the composition of trace-relations and property mappings. We describe the correctness theorem obtained for a multistep compiler, where each step is proved correct for a possibly different trace relation. We obtain the expected result: it is indeed possible to compose individual compiler correctness proofs for each step, to prove the correctness of the whole chain.

Assume \downarrow_i^s is a compilation chain from a source to an intermediate language and \downarrow_t^i from the intermediate to a target language.⁹ Let $\sim_{s,i} \subseteq \text{Traces} \times \text{Trace}_I$ and $\sim_{i,t} \subseteq \text{Trace}_I \times \text{Trace}_T$, such that $\downarrow_i^s \in \text{CC}^{\sim_{s,i}}$ and $\downarrow_t^i \in \text{CC}^{\sim_{i,t}}$. It is straightforward to show that compiling from source to target through $\downarrow_t^s = \downarrow_i^s; \downarrow_t^i$ satisfies the following

$$\text{CC}^{(\sim_{s,i}; \sim_{i,t})} \equiv \forall W \forall t. W \downarrow_t^s \rightsquigarrow t \Rightarrow \exists s \sim_{s,i}; \sim_{i,t} t. W \rightsquigarrow s$$

where $s \sim_{s,i}; \sim_{i,t} t \iff \exists w \in \text{Trace}_I. s \sim_{s,i} w \wedge w \sim_{i,t} t.$

It follows that target guarantees, as well as source obligations, can be established like in Section 2.2, using the composition of the two relation $\sim_{s,i}; \sim_{i,t}$, or just by composition of the property mappings. Similarly for $\downarrow_i^s \in \text{RTC}^{\sim_{s,i}}$ and $\downarrow_t^i \in \text{RTC}^{\sim_{i,t}}$, their composition $\downarrow_t^s \in \text{RTC}^{(\sim_{s,i}; \sim_{i,t})}$.

F In-Depth Trace-Relating Secure Compilation

We expand §5.3 and delve more in-depth on how to interpret related work as instance of our framework.

F.1 Trace-Relating Robust Preservation of Safety Properties

I/O events are not the only instance of events that compilers consider. Especially in the setting of secure compilation, where a compartmentalized partial program interacts with a context, *interaction traces* are often used [2, 29, 49, 53]. Consider a language analogous to that of the previous section, where the context C defines a set of functions F_c and the program defines a different set F_p . Interaction traces (generally) record the control flow of calls between these two sets via actions that are *call f v* and *ret v* [28]. These actions indicate a call to function f with parameter v and a return with return value v . In case the context calls a function in F_p (or returns to a function in F_p), the action is decorated with a $?$ (i.e., those actions are *call f v?* and *ret v?*). Dually, the program calling a function in F_c (or returning to it) generates an action decorated with a $!$ (i.e., those actions are *call f v!* and *ret v!*).

Patrignani and Garg [53] consider precisely such a setting. Moreover, they define a compiler that preserves safety properties of source programs (i.e., it is *RSC* in the sense of Figure 2) by relying on capability machines. The interesting point, however, is that they also consider source and target traces to be distinct since the source has **bools** and **nats** and the target only has **nats**.¹⁰ Thus, to prove *RSC*, they rely on a cross-language relation on values, which is scaled to trace actions, and then scaled point-wise to traces (analogously to what we have done in Sections 3.3, 3.4 and 5.2).

⁹ For the intermediate language we use a **verbatim**, **emerald** font.

¹⁰ Technically, their difference on values also encompasses heap locations and capabilities, but we elide that here since this is sufficient to draw our analogy.

Besides defining a relation on traces (which is an instance of \sim), they also define a relation between source and target safety properties that supports concurrent programs.¹¹ Thus, they really provide an instantiation of τ that maps all safe source traces to the related target ones. This ensures that no additional target trace is introduced in the target property, and source safety property are mapped to target safety ones by τ .

Their compiler is then proven to generate code that respects τ , so they achieve a variation of RSC^τ . Their proofs are based on standard techniques either for secure compilation (i.e., trace-based backtranslation [54]) and for correct compilation (i.e., forward/backward simulation [35]).

F.2 Trace-Relating Robust Preservation of Hypersafety Properties

Patrignani and Garg [52] study the preservation of hypersafety from the perspective of secure compilation. Again, their result can be interpreted in our setting. They consider reactive systems, where trace alphabets are partitioned in input actions $\alpha?$ and output actions $\alpha!$, whose concatenation generate traces $\overline{\alpha?}\alpha!$. We use the same notation as before and indicate such sequences as \mathbf{s} and \mathbf{t} respectively. The set of target output actions $\alpha!$ includes an action \checkmark that has no source counterpart (i.e., $\nexists\alpha? \sim \checkmark$), and whose output does not depend on internal state (thus it cannot leak secrets).¹² By emitting \checkmark whenever undesired inputs are fed to a compiled program (e.g., passing a **nat** when a **bool** is expected), hypersafety is preserved (as \checkmark does not leak secrets) [52].

More formally, they assume a relation on actions \sim that is total on the source actions and injective. From there, they define **TPC**—which here corresponds to an instance of τ —that maps the set of valid source traces to the set of valid target traces (that now mention \checkmark) as follows:

$$\mathbf{TPC}(\pi_S) = \left\{ \mathbf{t} \mid \mathbf{t} \in \bigcup_{n \in \mathbb{N}} \mathbf{int}_n(\pi_S) \right\} \text{ where } \mathbf{int}_0(\pi_S) = \{ \mathbf{t} \mid \exists \mathbf{s} \in \pi_S \wedge \mathbf{s} \sim \mathbf{t} \}$$

$\mathbf{int}_{n+1}(\pi_S) = \{ \mathbf{t} \mid (i) \mathbf{t} \equiv \mathbf{t}_1 \alpha? \checkmark \mathbf{t}_2 \wedge (ii) \mathbf{t}_1 \mathbf{t}_2 \in \mathbf{int}_n(\pi_S) \wedge (iii) \text{undesired}(\alpha?) \}$ where $\text{undesired}(\alpha?)$ indicates that $\alpha?$ is an undesired input (intuitively, this is an information that can be derived from the set of source traces [52]).

Informally, given a set of source traces π_S , **TPC** generates all target traces that are related (pointwise) to a source trace (case \mathbf{int}_0). Then (case \mathbf{int}_{n+1}), it adds all traces (\mathbf{t}) with interleavings of undesired input $\alpha?$ (point iii) followed by \checkmark (point i) as long as the interleavings split a trace $\mathbf{t}_1 \mathbf{t}_2$ that has already been mapped (point ii).

TPC is an instance of τ that maps source hypersafety to target hypersafety (and therefore, safety to safety), thus our theory can be instantiated for the preservation of these classes of hyperproperties as well.

¹¹ They call those safety properties monitors since they focus on safety [60] and indicate \mathbf{s} with \mathbf{M} and \mathbf{t} with \mathbf{M} .

¹² Technically, they assume a set of \checkmark actions, but for this analogy a single action suffices.

Bibliography

- [1] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. [A core calculus of dependency](#). *POPL*, 1999.
- [2] C. Abate, R. Blanco, D. Garg, C. Hrițcu, M. Patrignani, and J. Thibault. [Journey beyond full abstraction: Exploring robust property preservation for secure compilation](#). *CSF*, 2019.
- [3] A. Ahmed, D. Garg, C. Hrițcu, and F. Piessens. [Secure compilation \(Dagstuhl Seminar 18201\)](#). *Dagstuhl Reports*, 8(5), 2018.
- [4] A. Anand, A. Appel, G. Morrisett, Z. Paraskevopoulou, R. Pollack, O. S. Belanger, M. Sozeau, and M. Weaver. [CertiCoq: A verified compiler for Coq](#). *CoqPL Workshop*, 2017.
- [5] K. Backhouse and R. Backhouse. [Safety of abstract interpretations for free, via logical relations and Galois connections](#). *Science of Computer Programming*, 51(1-2), 2004.
- [6] G. Barthe, B. Grégoire, and V. Laporte. [Secure compilation of side-channel countermeasures: the case of cryptographic “constant-time”](#). *CSF*, 2018.
- [7] L. Beringer, G. Stewart, R. Dockins, and A. W. Appel. [Verified compilation for shared-memory C](#). *ESOP*, 2014.
- [8] F. Besson, S. Blazy, and P. Wilke. [A verified CompCert front-end for a memory model supporting pointer arithmetic and uninitialised data](#). *Journal of Automated Reasoning*, 62(4), 2019.
- [9] S. Boldo, J. Jourdan, X. Leroy, and G. Melquiond. [Verified compilation of floating-point computations](#). *Journal of Automated Reasoning*, 54(2), 2015.
- [10] M. Busi, P. Degano, and L. Galletta. [Translation validation for security properties](#). *CoRR*, abs/1901.05082, 2019.
- [11] Q. Cao, L. Beringer, S. Gruetter, J. Dodds, and A. W. Appel. [VST-Floyd: A separation logic tool to verify correctness of C programs](#). *Journal of Automated Reasoning*, 61(1-4), 2018.
- [12] Q. Carbonneaux, J. Hoffmann, T. Ramanand, and Z. Shao. [End-to-end verification of stack-space bounds for C programs](#). *PLDI*, 2014.
- [13] C. Cimpanu. [Microsoft: 70 percent of all security bugs are memory safety issues](#). *ZDNet*, 2019.
- [14] M. R. Clarkson and F. B. Schneider. [Hyperproperties](#). *JCS*, 18(6), 2010.
- [15] P. Cousot. [Constructive design of a hierarchy of semantics of a transition system by abstract interpretation](#). *TCS*, 277(1-2), 2002.
- [16] P. Cousot and R. Cousot. [Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints](#). *POPL*, 1977.
- [17] V. D’Silva, M. Payer, and D. X. Song. [The correctness-security gap in compiler optimization](#). *S&P Workshops*, 2015.
- [18] J. Engelfriet. [Determinacy implies \(observation equivalence = trace equivalence\)](#). *TCS*, 36, 1985.
- [19] R. Focardi and R. Gorrieri. [A taxonomy of security properties for process algebras](#). *JCS*, 3(1), 1995.
- [20] P. H. Gardiner, C. E. Martin, and O. De Moor. [An algebraic construction of predicate transformers](#). *Science of Computer Programming*, 22(1-2), 1994.
- [21] R. Giacobazzi and I. Mastroeni. [Abstract non-interference: a unifying framework for weakening information-flow](#). *ACM Transactions on Privacy and Security*, 21(2), 2018.
- [22] J. A. Goguen and J. Meseguer. [Security policies and security models](#). *S&P*, 1982.
- [23] R. Gu, Z. Shao, J. Kim, X. N. Wu, J. Koenig, V. Sjöberg, H. Chen, D. Costanzo, and T. Ramanand. [Certified concurrent abstraction layers](#). *PLDI*, 2018.

- [24] I. Haller, Y. Jeon, H. Peng, M. Payer, C. Giuffrida, H. Bos, and E. van der Kouwe. [TypeSan: Practical type confusion detection](#). *CCS*, 2016.
- [25] Heartbleed. [The Heartbleed bug](http://heartbleed.com/). <http://heartbleed.com/>, 2014.
- [26] C. Hrițcu, D. Chisnall, D. Garg, and M. Payer. [Secure compilation](#). SIGPLAN PL Perspectives Blog, 2019.
- [27] C. Hur and D. Dreyer. [A Kripke logical relation between ML and assembly](#). *POPL*, 2011.
- [28] A. Jeffrey and J. Rathke. [Java Jr: Fully abstract trace semantics for a core Java language](#). *ESOP*, 2005.
- [29] Y. Juglaret, C. Hrițcu, A. Azevedo de Amorim, B. Eng, and B. C. Pierce. [Beyond good and evil: Formalizing the security guarantees of compartmentalizing compilation](#). *CSF*, 2016.
- [30] J. Kang, C. Hur, W. Mansky, D. Garbuzov, S. Zdancewic, and V. Vafeiadis. [A formal C memory model supporting integer-pointer casts](#). *PLDI*, 2015.
- [31] J. Kang, Y. Kim, C.-K. Hur, D. Dreyer, and V. Vafeiadis. [Lightweight verification of separate compilation](#). *POPL*, 2016.
- [32] L. Lamport and F. B. Schneider. [Formal foundation for specification and verification](#). In *Distributed Systems: Methods and Tools for Specification, An Advanced Course*, 1984.
- [33] C. Lattner. [What every C programmer should know about undefined behavior #1/3](#). LLVM Project Blog, 2011.
- [34] X. Leroy. [Formal verification of a realistic compiler](#). *CACM*, 52(7), 2009.
- [35] X. Leroy. [A formally verified compiler back-end](#). *JAR*, 43(4), 2009.
- [36] X. Leroy. [The formal verification of compilers \(DeepSpec Summer School 2017\)](#), 2017.
- [37] I. Mastroeni and M. Pasqua. [Verifying bounded subset-closed hyperproperties](#). *SAS*, 2018.
- [38] J. McCarthy and J. Painter. [Correctness of a compiler for arithmetic expressions](#). *Mathematical Aspects Of Computer Science 1*, 19 of Proceedings of Symposia in Applied Mathematics, 1967.
- [39] A. Melton, D. A. Schmidt, and G. E. Strecker. [Galois connections and computer science applications](#). In *Proceedings of a Tutorial and Workshop on Category Theory and Computer Programming*, 1986.
- [40] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, Berlin, Heidelberg, 1982.
- [41] R. Milner and R. Weyhrauch. [Proving compiler correctness in a mechanized logic](#). In *Proceedings of 7th Annual Machine Intelligence Workshop, volume 7 of Machine Intelligence*, 1972.
- [42] F. L. Morris. [Advice on structuring compilers and proving them correct](#). *POPL*, 1973.
- [43] E. Mullen, D. Zuniga, Z. Tatlock, and D. Grossman. [Verified peephole optimizations for CompCert](#). *PLDI*, 2016.
- [44] D. A. Naumann. [A categorical model for higher order imperative programming](#). *Mathematical Structures in Computer Science*, 8(4), 1998.
- [45] D. A. Naumann and M. Ngo. [Whither specifications as programs](#). In *International Symposium on Unifying Theories of Programming*. Springer, 2019.
- [46] G. Neis, C. Hur, J. Kaiser, C. McLaughlin, D. Dreyer, and V. Vafeiadis. [Pilsner: a compositionally verified compiler for a higher-order imperative language](#). *ICFP*, 2015.
- [47] M. New, W. J. Bowman, and A. Ahmed. [Fully abstract compilation via universal embedding](#). In *ICFP'16*, 2016.
- [48] M. Pasqua and I. Mastroeni. [On topologies for \(hyper\)properties](#). *CEUR*, 2017.
- [49] M. Patrignani. *The Tome of Secure Compilation: Fully Abstract Compilation to Protected Modules Architectures*. PhD thesis, KU Leuven, Leuven, Belgium, 2015.
- [50] M. Patrignani. [Why should anyone use colours? or, syntax highlighting beyond code snippets](#), 2020.
- [51] M. Patrignani and D. Clarke. [Fully abstract trace semantics for protected module architectures](#). *Computer Languages, Systems & Structures*, 42, 2015.

- [52] M. Patrignani and D. Garg. Secure compilation and hyperproperty preservation. *CSF*, 2017.
- [53] M. Patrignani and D. Garg. Robustly safe compilation. *ESOP*, 2019.
- [54] M. Patrignani, A. Ahmed, and D. Clarke. Formal approaches to secure compilation: A survey of fully abstract compilation and related work. *ACM Computing Surveys*, 2019.
- [55] D. Patterson and A. Ahmed. The next 700 compiler correctness theorems (functional pearl). *PACMPL*, 3(ICFP), 2019.
- [56] T. Ramananandro, Z. Shao, S. Weng, J. Koenig, and Y. Fu. A compositional semantics for verified separate compilation and linking. *CPP*, 2015.
- [57] J. Regehr. A guide to undefined behavior in C and C++, part 3. Embedded in Academia blog, 2010.
- [58] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. *CSFW*, 2005.
- [59] A. Sabry and P. Wadler. A reflection on call-by-value. *ACM Transactions on Programming Languages and Systems*, 19(6), 1997.
- [60] F. B. Schneider. Enforceable security policies. *TISSEC*, 3(1), 2000.
- [61] J. Sevcík, V. Vafeiadis, F. Z. Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3), 2013.
- [62] L. Skorstengaard, D. Devriese, and L. Birkedal. StkTokens: Enforcing Well-bracketed Control Flow and Stack Encapsulation Using Linear Capabilities. *Proc. ACM Program. Lang.*, 3(POPL), 2019.
- [63] G. Stewart, L. Beringer, S. Cuellar, and A. W. Appel. Compositional CompCert. *POPL*, 2015.
- [64] Y. K. Tan, M. O. Myreen, R. Kumar, A. Fox, S. Owens, and M. Norrish. The verified CakeML compiler backend. *Journal of Functional Programming*, 29, 2019.
- [65] X. Wang, H. Chen, A. Cheung, Z. Jia, N. Zeldovich, and M. F. Kaashoek. Undefined behavior: What happened to my code? *APSYS*, 2012.
- [66] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. *SOSP*, 2013.
- [67] Y. Wang, P. Wilke, and Z. Shao. An abstract stack based approach to verified compositional compilation to machine code. *PACMPL*, 3(POPL), 2019.
- [68] L. Xia, Y. Zakowski, P. He, C. Hur, G. Malecha, B. C. Pierce, and S. Zdancewic. Interaction trees: representing recursive and impure programs in Coq. *PACMPL*, 4(POPL), 2020.
- [69] A. Zakinthinos and E. S. Lee. A general theory of security properties. *S&P*, 1997.
- [70] J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. *POPL*, 2012.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

