An Extended Account of Trace-Relating Compiler Correctness and Secure Compilation

CARMINE ABATE[*], MPI-SP, Germany
ROBERTO BLANCO[*], MPI-SP, Germany
ȘTEFAN CIOBÂCĂ, Alexandru Ioan Cuza University Iași, Romania
ADRIEN DURIER[*], MPI-SP, Germany
DEEPAK GARG, Max Planck Institute for Software Systems, Germany
CĂTĂLIN HRIȚCU[*], MPI-SP, Germany
MARCO PATRIGNANI, Stanford, USA and CISPA Helmholz Center for Information Security, Germany
ÉRIC TANTER[*], University of Chile, Chile
JÉRÉMY THIBAULT[*], MPI-SP, Germany

Compiler correctness, in its simplest form, is defined as the inclusion of the set of traces of the compiled program in the set of traces of the original program. This is equivalent to the preservation of all trace properties. Here, traces collect, for instance, the externally observable events of each execution. However, this definition requires the set of traces of the source and target languages to be 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

[*] Part of this work was conducted while these authors were employed at or visiting Inria Paris

This paper revises and extends the work of ?] presented at ESOP'20 with the following additions. It contains a more complete account of the classes of properties that can be preserved by correct compilers by discussing safety and hyperproperty preservation. It discusses recent work on the preservation of noninterference through compilation [? ? ?] and interprets this work within the presented framework. It unifies the language presentation for the compilers that are proven correct using different relations. It provides a self-contained and in-depth analysis of the classes of properties that can be preserved by secure compilers by discussing subset-closed hyperproperties, hypersafety, 2-relational properties, 2-relational safety and 2-relational hyperproperties. Generally, this paper provides more intuition and explanation for some of the presented notions as well as for the discussed instances of our theory.

Please note that Coq symbols as well as Compiler Criteria are links: the former to Coq files in the external repository https://github.com/secure-compilation/different_traces, the latter to definitions inside this document.

Authors' addresses: Carmine Abate, MPI-SP, Germany, carmine.abate@mpi-sp.org; Roberto Blanco, MPI-SP, Germany, roberto.blanco@mpi-sp.org; Ștefan Ciobâcă, Department of Computer Science, Alexandru Ioan Cuza University Iași, Romania, stefan.ciobaca@info.uaic.ro; Adrien Durier, MPI-SP, Germany, adrien.durier@mpi-sp.org; Deepak Garg, Max Planck Institute for Software Systems, Saarland Informatics Campus, Germany, dg@mpi-sws.org; Cătălin Hrițcu, MPI-SP, Germany, catalin.hritcu@gmail.com; Marco Patrignani, Stanford, USA , CISPA Helmholz Center for Information Security, Germany, mp@cs.stanford.edu; Éric Tanter, Computer Science Department (DCC), University of Chile, Chile, etanter@dcc.uchile.cl; Jérémy Thibault, MPI-SP, Germany, jeremy.thibault@mpi-sp.org.

0164-0925/2021/1-ART1
https://doi.org/10.1145/3460860

applies to many definitions of *secure* compilation, which characterize the protection of a compiled program linked against adversarial code.

CCS Concepts: • **Security and privacy** → **Formal security models**; • **Software and its engineering** → **Compilers**; *Software verification*;

Additional Key Words and Phrases: trace properties, hyperproperties, property-preserving compilation, compiler correctness, secure compilation, Galois connection, formal languages, programming languages

ACM Reference Format:
Carmine Abate, Roberto Blanco, Ștefan Ciobâcă, Adrien Durier, Deepak Garg, Cătălin Hrițcu, Marco Patrignani, Éric Tanter, and Jérémy Thibault. 2021. An Extended Account of Trace-Relating Compiler Correctness and Secure Compilation. *ACM Trans. Program. Lang. Syst.* 1, 1, Article 1 (January 2021), 44 pages. https://doi.org/10.1145/3460860

1 INTRODUCTION

Compiler correctness is an old idea [? ? ?] that has seen a significant revival in recent times. This new wave was started by the creation of the CompCert verified C compiler [?] and continued by the proposal of many significant extensions and variants of CompCert [? ? ? ? ? ? ? ? ? ? ?] and the success of many other milestone compiler verification projects, including Vellvm [?], Pilsner [?], CakeML [?], and CertiCoq [?]. Verification through proof assistants allows the user of a compiler to trust the proofs without diving into all of the details. Still, in order to clearly understand the benefits and limitations of using a verified compiler, she has to deeply understand the statement of correctness. This is true not just for correct compilation, but also for secure compilation, which is the more recent idea that a compilation chain should not just provide correctness but also security against co-linked adversarial components [? ?].

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 [?] and CakeML [?], 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 inclusion of the set of traces of the compiled program in the set of traces of the original program. Formally [?]:

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

$$\forall \mathsf{W}\ t.\ \mathsf{W}{\downarrow} \rightsquigarrow t \Rightarrow \mathsf{W} \rightsquigarrow t$$

This definition says that for any whole[1] source program W, if we compile it (denoted $\mathsf{W}{\downarrow}$), execute it in the semantics of the target language, and observe a trace t, then the original W can produce *the same* trace t in 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. Definition 1.1 implicitly assumes that the source and target traces are drawn from the very same set, and requires that any target trace produced by a compiled program can be faithfully reproduced by the source program. In practice, existing verified compiler adopt a less restrictive formulation of compiler correctness:

[1] For simplicity, for now we ignore separate compilation and linking, returning to it in §6.

[2] Typesetting convention [?]: 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.

CompCert [?] The original compiler correctness theorem of CompCert [?] can be seen as an instance of basic compiler correctness, but it does not provide any guarantees for programs that can exhibit undefined behavior [?]. 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 [? ? ?] and serious confusion even among experienced C and C++ developers [? ? ? ?]. As such, since 2010, CompCert provides an additional top-level correctness theorem[3] that better accounts for the presence of unsafe programs by providing guarantees for them up to the point when they encounter undefined behavior [?]. 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 [?] 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.

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 $\mathsf{Trace}_\mathsf{S}$ and $\mathsf{Trace}_\mathsf{T}$, and (b) being parameterized by an arbitrary *trace relation* $\sim$.

Definition 1.2 (Trace-Relating Compiler Correctness ($\mathsf{CC}^\sim$)). A compiler $\downarrow$ is *correct* with respect to a trace relation $\sim \subseteq \mathsf{Trace}_\mathsf{S} \times \mathsf{Trace}_\mathsf{T}$ iff

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

This definition requires that, for any target trace t produced by the compiled program $\mathsf{W}{\downarrow}$, 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., $\mathsf{s} \sim \mathsf{t}$). By choosing the trace relation appropriately, one can recover the different notions of compiler correctness presented above:

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

CompCert Undefined behavior is modeled in CompCert as a trace-terminating event *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 $\mathsf{Trace}_\mathsf{S} = \mathsf{Trace}_\mathsf{T}$. Nevertheless, the relation between source and target traces with which to instantiate $\mathsf{CC}^\sim$ to obtain CompCert's current theorem is the following (note that we denote *finite* traces–or prefixes– as m)

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

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

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

$$\mathsf{s} \sim \mathsf{t} \ \equiv\ \mathsf{s} = \mathsf{t} \vee (\exists m.\ m \leq \mathsf{s}.\ \mathsf{t} = m \cdot \mathsf{Resource\_limit\_hit})$$

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

> The resulting $\mathsf{CC}^{\sim}$ instance relates a target trace ending in $\mathsf{Resource\_limit\_hit}$ after executing prefix 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 $\mathsf{CC}^{\sim}$: 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 $\mathsf{CC}^{\sim}$ than for $\mathsf{CC}^{=}$. 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 $\mathsf{CC}^{\sim}$ 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:

$$\mathsf{s} \sim \mathsf{t} \;\equiv\; \exists \mathsf{W}.\mathsf{W} \rightsquigarrow \mathsf{s} \wedge \mathsf{W}\downarrow \rightsquigarrow \mathsf{t}$$

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

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

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 $\mathsf{CC}^{\sim}$ 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 $\mathsf{CC}^{\sim}$ compiler?

Far from being mere hypothetical questions, they can help the developer of a verified compiler 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 $\mathsf{CC}^{\sim}$. Building upon a bijection between relations and Galois connections [**? ? ?**], 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"):

$$\tilde{\tau}(\pi_{\mathsf{S}}) = \{\mathsf{t} \mid \exists \mathsf{s}.\ \mathsf{s} \sim \mathsf{t} \wedge \mathsf{s} \in \pi_{\mathsf{S}}\}$$
$$\tilde{\sigma}(\pi_{\mathsf{T}}) = \{\mathsf{s} \mid \forall \mathsf{t}.\ \mathsf{s} \sim \mathsf{t} \Rightarrow \mathsf{t} \in \pi_{\mathsf{T}}\}$$

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):

- $\mathsf{TP}^{\tilde{\tau}}$ quantifies over all source trace properties and uses $\tilde{\tau}$ to obtain the corresponding target properties;

- $\mathsf{TP}^{\tilde{\sigma}}$ 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 $\mathsf{CC}^{\sim}$, yielding a novel trinitarian view of compiler correctness (Figure 1).

$$\begin{tikzcd}
 & & \forall \mathsf{W}.\ \forall \mathsf{t}.\ \mathsf{W}\downarrow \rightsquigarrow \mathsf{t} \Rightarrow \exists \mathsf{s} \sim \mathsf{t}.\ \mathsf{W} \rightsquigarrow \mathsf{s} & & \\
 & & \mathsf{CC}^{\sim} \arrow[u, phantom, "\equiv"] \arrow[dr, Leftrightarrow] & & \\
\begin{array}{l} \forall \pi_{\mathsf{T}}.\ \forall \mathsf{W}.\ \mathsf{W} \models \tilde{\sigma}(\pi_{\mathsf{T}}) \\ \Rightarrow \mathsf{W}\downarrow \models \pi_{\mathsf{T}} \end{array} \arrow[r, phantom, "\equiv"] & \mathsf{TP}^{\tilde{\sigma}} \arrow[ur, Leftrightarrow] \arrow[rr, Leftrightarrow] & & \mathsf{TP}^{\tilde{\tau}} \arrow[r, phantom, "\equiv"] & \begin{array}{l} \forall \pi_{\mathsf{S}}.\ \forall \mathsf{W}.\ \mathsf{W} \models \pi_{\mathsf{S}} \\ \Rightarrow \mathsf{W}\downarrow \models \tilde{\tau}(\pi_{\mathsf{S}}) \end{array}
\end{tikzcd}$$

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 relations between source and target traces. While, as discussed above, specific instances of the $\mathsf{CC}^{\sim}$ definition have already been used in practice, we seem to be the first to propose assessing the meaningfulness of $\mathsf{CC}^{\sim}$ 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 $\mathsf{CC}^{\sim}$, $\mathsf{TP}^{\tilde{\sigma}}$, and $\mathsf{TP}^{\tilde{\tau}}$ 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 [?], an analogous equivalence holds.
- We extend these results from the preservation of trace properties to the larger class of subset-closed hyperproperties, e.g., noninterference (§3.1)[4], and to the classes of safety properties (§3.2) and all hyperproperties (§3.3).
- We use $\mathsf{CC}^{\sim}$ compilers of various complexities to illustrate that our view on compiler correctness naturally accounts for undefined behavior (§4.1), resource exhaustion (§4.2), different source and target values (§4.3), and differences in the granularity of data and observable events (§4.4). We expect these ideas to extend to 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 [?], 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 (§5). We formalize the guarantee obtained by correct compilation of a noninterfering program as *abstract noninterference* [?], a weakening of target noninterference. Dually, we identify a family of declassifications of target noninterference for which source reasoning is possible.

[4] Given the deterministic nature of our programs, we consider notions of noninterference that are often used in deterministic languages. We leave notions of noninterference in nondeterministic languages for future work.

- We show that the trinitarian view also extends to a large class of *secure compilation* definitions [?], formally characterizing the protection of the compiled program against linked adversarial code (§6). For each secure compilation definition we again propose both a property-free characterization in the style of $\mathrm{CC}^{\sim}$, 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 [?].
- We provide instances of secure compilers that preserve three different classes of hyperproperties (trace, safety and hypersafety properties) when targeting a language with additional trace events that are not possible in the source (§7).

The results and insights that we provide often follow one's expected intuition and may be considered unsurprising. However our framework is the first to capture such expectations formally and precisely, and as such it provides a uniform way to discuss these and to formalise future (possibly surprising) ones. The paper closes with discussions of related (§8) and future work (§9). Some technical proofs can be found in the appendix (§A).

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 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., $\mathrm{CC}^{=}$) is enough to preserve not just trace properties but also all subset-closed hyperproperties [?]. For this, we show that $\mathrm{CC}^{\sim}$ 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 $\tilde{\tau}$ (§3.1).

2.1 Property Mappings

As explained in §1, trace-relating compiler correctness $\mathrm{CC}^{\sim}$, 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 focus on trace properties of the source (and then interpret them in the target), or on trace properties of the target (and then interpret them in the source). Formally we need two property mappings, $\tau : 2^{\mathrm{Trace}_S} \to 2^{\mathrm{Trace}_T}$ and $\sigma : 2^{\mathrm{Trace}_T} \to 2^{\mathrm{Trace}_S}$, which lead us to the following generalization of trace property preservation (TP).

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

$$\mathrm{TP}^{\tau} \equiv \forall \pi_S.\ \forall W.\ W \models \pi_S \Rightarrow W\downarrow \models \tau(\pi_S)$$

$$\mathsf{TP}^{\sigma} \equiv \forall \pi_{\mathsf{T}}.\, \forall \mathsf{W}.\, \mathsf{W} \models \sigma(\pi_{\mathsf{T}}) \Rightarrow \mathsf{W}{\downarrow} \models \pi_{\mathsf{T}}$$

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

i) Given π_{T}, the target interpretation of the source obligation $\sigma(\pi_{\mathsf{T}})$ should actually guarantee that π_{T} holds, i.e., $\tau(\sigma(\pi_{\mathsf{T}})) \subseteq \pi_{\mathsf{T}}$;

ii) Dually for π_{S}, we would not want the source obligation for $\tau(\pi_{\mathsf{S}})$ to be harder than π_{S} itself, i.e., $\sigma(\tau(\pi_{\mathsf{S}})) \supseteq \pi_{\mathsf{S}}$.

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 [**? ?**].

Definition 2.2 (Galois connection)**.** Let $(X, \preceq)$ and $(Y, \sqsubseteq)$ be two posets. A pair of maps, $\alpha : X \to Y$, $\gamma : Y \to X$ 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 $\alpha : (X, \preceq) \leftrightarrows (Y, \sqsubseteq) : \gamma$ to denote a Galois connection, or simply $\alpha : X \leftrightarrows Y : \gamma$, or even $\alpha \leftrightarrows \gamma$ when the involved posets are clear from context.

Lemma 2.3 (Characteristic property of Galois connections)**.** If $\alpha{:}(X, \preceq) \leftrightarrows (Y, \sqsubseteq){:}\gamma$ is a Galois connection, then α, γ are monotone and $id \preceq \gamma \circ \alpha$ and $\alpha \circ \gamma \sqsubseteq id$, i.e.,

$$\forall x \in X.\, x \preceq \gamma(\alpha(x))$$
$$\forall y \in Y.\, \alpha(\gamma(y)) \sqsubseteq y$$

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 conditions *i*), *ii*) above, i.e., $\tau(\sigma(\pi_{\mathsf{T}})) \subseteq \pi_{\mathsf{T}}$ and $\sigma(\tau(\pi_{\mathsf{S}})) \supseteq \pi_{\mathsf{S}}$.[5] These 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^{τ} and TP^{σ} coincide 🐓)**.** Let $\tau : 2^{\mathsf{Trace}_{\mathsf{S}}} \rightleftarrows 2^{\mathsf{Trace}_{\mathsf{T}}} : \sigma$ be a Galois connection, with τ and σ the lower and upper adjoints (resp.). Then $\mathsf{TP}^{\tau} \iff \mathsf{TP}^{\sigma}$.

Proof. Notice that if a program satisfies a property π, then it satisfies every less restrictive i.e., bigger property $\pi' \supseteq \pi$. Building on this:

($\Rightarrow$) Assume TP^{τ} and that W satisfies $\sigma(\pi_{\mathsf{T}})$. Apply TP^{τ} to W and $\sigma(\pi_{\mathsf{T}})$ and deduce that $\mathsf{W}{\downarrow}$ satisfies $\tau(\sigma(\pi_{\mathsf{T}})) \subseteq \pi_{\mathsf{T}}$.

($\Leftarrow$) Assume TP^{σ} and that W satisfies $\pi_{\mathsf{S}} \subseteq \sigma(\tau(\pi_{\mathsf{S}}))$. Apply TP^{σ} to W and $\sigma(\tau(\pi_{\mathsf{S}}))$ deducing $\mathsf{W}{\downarrow}$ satisfies $\tau(\pi_{\mathsf{S}})$.

□

2.2 Trace Relations and Property Mappings

We now investigate the relation between $\mathsf{CC}^{\sim}$, 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 the design of a correct compiler. For a $\mathsf{CC}^{\sim}$ compiler, the equivalence makes explicit both the guarantees one has after compilation

[5] While target traces are often *"more concrete"* than source ones, trace properties 2^{Trace} (which in Coq we represent as the function type Trace→Prop) are contravariant in Trace and thus target properties correspond to the *abstract domain*.

($\tilde{\tau}$) 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, $\mathsf{CC}^{\sim}$, for which more convenient proof techniques exist in the literature [**?** **?**].

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

$$\tilde{\tau} = \lambda\, \pi \in 2^X.\ \{y \mid \exists x.\, x \sim y \wedge x \in \pi\}$$

$$\tilde{\sigma} = \lambda\, \pi \in 2^Y.\ \{x \mid \forall y.\, x \sim y \Rightarrow y \in \pi\}$$

When trace relations are considered, the corresponding 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 🐓). For any trace relation $\sim$ and its existential and universal images $\tilde{\tau}$ and $\tilde{\sigma}$, we have:

$$\begin{tikzcd}
& \mathsf{CC}^{\sim} \arrow[dl, Leftrightarrow] \arrow[dr, Leftrightarrow] & \\
\mathsf{TP}^{\tilde{\sigma}} \arrow[rr, Leftrightarrow] & & \mathsf{TP}^{\tilde{\tau}}
\end{tikzcd}$$

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 (🐓). The theorem can be stated in a slightly more general form (Theorem 2.8), exploiting an isomorphism between the category of sets and relations and a sub category of monotonic predicate transformers [**?**]. We specialize this isomorphism to what is of interest for our purposes and deduce a bijective correspondence between trace relations and Galois connections on properties.

Lemma 2.7 (Trace relations ≅ Galois connections on trace properties). The function $\sim\ \mapsto \tilde{\tau} \leftrightarrows \tilde{\sigma}$ that maps a trace relation to its existential and universal images is a bijection between trace relations $2^{\mathsf{Trace_S} \times \mathsf{Trace_T}}$ and Galois connections on trace properties $2^{\mathsf{Trace_S}} \leftrightarrows 2^{\mathsf{Trace_T}}$. Its inverse is $\tau \leftrightarrows \sigma \mapsto \hat{\sim}$, where $\mathsf{s} \mathrel{\hat{\sim}} \mathsf{t} \equiv \mathsf{t} \in \tau(\{\mathsf{s}\})$.

The bijection just introduced allows us to generalize Theorem 2.6 and switch anytime between the three views of compiler correctness described earlier.

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

Note 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 (§4.2 presents an example).

3 PRESERVING OTHER (HYPER)PROPERTY CLASSES

In this section we investigate how to preserve other classes of (hyper)properties beyond trace properties: subset-closed hyperproperties (§3.1) safety properties (§3.2) and arbitrary hyperproperties that are not just subset-closed (§3.3). 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. For subset-closed hyperproperties we have to refine the Galois connection to ensure the information "$\mathsf{H_S}$ is subset-closed" is not lost with the application of $\tilde{\tau}$. Similarly, when looking at safety properties, we have to preserve the information that a propery is a safety property. For arbitrary hyperproperties one might instead require that no information at all is lost during the (pre or post) composition of $\tilde{\tau}$ and $\tilde{\sigma}$. The section concludes with a comparison of the criteria in terms of relative strengths (§3.4).

3.1 Preservation of Subset-Closed Hyperproperties

Hyperproperty preservation is a strong requirement in general. Fortunately, many interesting hyperproperties are *subset-closed* (*SCH* for short) (e.g., noninterference), and these are known to be preserved by refinement [?]. When the trace semantics is common to source and target languages, a subset-closed hyperproperty is preserved if the behaviors of the compiled program refine the behaviors of the source program, which coincides with the statement of $\mathsf{CC}^{=}$. We generalize this result to the trace-relating setting, introducing two other equivalent characterizations of $\mathsf{CC}^{\sim}$ in terms of preservation of subset-closed hyperproperties (Theorem 3.3). In order to do so we close under subsets the images of both $\tilde{\tau}$ and $\tilde{\sigma}$ so that source subset-closed hyperproperties are mapped to target subset-closed ones and viceversa.

First, a hyperproperty H is defined as a set of sets of traces, $H \in 2^{2^{\mathrm{Trace}}}$ (recall that *Traces* is the set of all traces) [?]. 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.

Definition 3.1 (Hyperproperty Satisfaction [?]). A program W satisfies a hyperproperty H, written $W \models H$,[6] iff $beh(W) \in H$, where $beh(W) = \{t \mid W \rightsquigarrow t\}$.

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 3.2 (Lifting property mappings to hyperproperty mappings). Let $\tau : 2^{\mathsf{Trace_S}} \to 2^{\mathbf{Trace_T}}$ and $\sigma : 2^{\mathbf{Trace_T}} \to 2^{\mathsf{Trace_S}}$ be arbitrary property mappings. The images of $\mathsf{H_S} \in 2^{2^{\mathsf{Trace_S}}}$, $\mathbf{H_T} \in 2^{2^{\mathbf{Trace_T}}}$ under τ and σ are, respectively:

$$\tilde{\tau}(\mathsf{H_S}) = \{\tau(\pi_{\mathsf{S}}) \mid \pi_{\mathsf{S}} \in \mathsf{H_S}\} \qquad\qquad \tilde{\sigma}(\mathbf{H_T}) = \{\sigma(\boldsymbol{\pi}_{\mathbf{T}}) \mid \boldsymbol{\pi}_{\mathbf{T}} \in \mathbf{H_T}\}$$

Formally, we are defining two new mappings, this time on hyperproperties, but with a small abuse of notation we still denote them $\tilde{\tau}$ and $\tilde{\sigma}$.

Interestingly, it is not possible to apply the argument used for $\mathsf{CC}^{=}$ to show that a $\mathsf{CC}^{\sim}$ compiler guarantees $\mathsf{W}{\downarrow} \models \tilde{\tau}(\mathsf{H_S})$ whenever $\mathsf{W} \models \mathsf{H_S}$. This is because direct images do not necessarily preserve subset-closure [? ?].[JT: not a sentence?] We therefore close the image of $\tilde{\tau}$ and $\tilde{\sigma}$ under subsets (denoted as $Cl_{\subseteq}$) and obtain the following result:

Theorem 3.3 (Preservation of Subset-Closed Hyperproperties 🐓). For any trace relation $\sim$ and its existential and universal images lifted to hyperproperties, $\tilde{\tau}$ and $\tilde{\sigma}$, and for $Cl_{\subseteq}(H) = \{\pi \mid \exists \pi' \in H.\, \pi \subseteq \pi'\}$, we have the following:

$$\mathsf{SCHP}^{Cl_{\subseteq}\circ\tilde{\tau}} \equiv \forall \mathsf{W} \forall \mathsf{H_S} \in \mathsf{SCH_S}. \mathsf{W} \models \mathsf{H_S} \Rightarrow \mathsf{W}{\downarrow} \models Cl_{\subseteq}(\tilde{\tau}(\mathsf{H_S}));$$

$$\mathsf{SCHP}^{Cl_{\subseteq}\circ\tilde{\sigma}} \equiv \forall \mathsf{W} \forall \mathbf{H_T} \in \mathbf{SCH_T}. \mathsf{W} \models Cl_{\subseteq}(\tilde{\sigma}(\mathbf{H_T})) \Rightarrow \mathsf{W}{\downarrow} \models \mathbf{H_T}.$$

$$\begin{tikzcd}
 & \mathsf{CC}^{\sim} & \\
\mathsf{SCHP}^{Cl_{\subseteq}\circ\tilde{\tau}} \arrow[ur, Leftrightarrow] \arrow[rr, Leftrightarrow] & & \mathsf{SCHP}^{Cl_{\subseteq}\circ\tilde{\sigma}} \arrow[ul, Leftrightarrow]
\end{tikzcd}$$

The use of $Cl_{\subseteq}$ in Theorem 3.3 implies a loss of precision in preserving subset-closed hyperproperties through compilation. In §5, we focus on a specific security-relevant subset-closed hyperproperty, noninterference, and show that such a loss of precision can be seen as a declassification. Instead, now we define the trinity and the related formal machinery for safety properties preservation.

[6] In case of ambiguity with property satisfaction the class of H will be made explicit.

3.2 Preserving Safety Properties

The class of *Safety* properties collects all trace properties prescribing that *"something bad never happens"* or equivalently, all trace properties whose violation can be monitored and, once observed, no longer restored [?]. More abstractly safety properties can be defined as the closed sets of a topology [? ?], with no need to consider any particular structure on the traces. To ease the presentation, we consider the trace model adopted by ?] where traces resemble lists and streams of events. This model naturally comes with a notion of *prefixes* and a relation between a prefix m and a trace t, written $m \leq t$. Intuitively, π is a safety property if any trace t *violating* the property extends a "bad prefix" m that witnesses such a violation. Every safety property is therefore uniquely defined by the set of its "bad prefixes". We recall below the definition and the characterization of safety properties in terms of sets of finite prefixes m.

Definition 3.4 (Safety Properties [?]). Let π be a trace property. Then,

$\pi \in \mathit{Safety}$ *iff* $\forall t \notin \pi.\ \exists m \leq t.\ \forall t'.\ m \leq t' \Rightarrow t' \notin \pi.$

Equivalently, $\pi \in \mathit{Safety}$ *iff* there exists a set of finite prefixes M, such that

$$\forall t.\ t \notin \pi \iff (\exists m \in M.\ m \leq t)$$

Due to this characterization of safety properties through finite prefixes (Definition 3.4), the preservation of all and only the safety properties is equivalent to $\mathsf{CC}^{=}$ restricted to finite prefixes. [CA: I don't have a reference for this, but it is folklore and immediate to proof. – If I have time I will add a paper proof in the appendix]

Theorem 3.5. The following are equivalent:

$$\mathsf{SC}^{=} \equiv \forall \mathsf{W}, m.\ \mathsf{W}\!\downarrow \rightsquigarrow^{*} m \Rightarrow \mathsf{W} \rightsquigarrow^{*} m$$
$$\mathsf{SP} \equiv \forall \pi \in \mathit{Safety}.\ \mathsf{W} \models \pi \Rightarrow \mathsf{W}\!\downarrow \models \pi$$

where $W \rightsquigarrow^{*} m$ stands for $\exists t.\ m \leq t \wedge W \rightsquigarrow t$.

Unfolding $\rightsquigarrow^{*}$ we can interpret $\mathsf{SC}^{=}$ as follows. Whenever $\mathsf{W}\!\downarrow$ produces a trace $t \geq m$ that violates a specific safety property, namely, the one defined by the singleton prefix set $\{m\}$, then W violates the *same* safety property, producing a trace $t' \geq m$ but possibly distinct from t.
The generalization we propose of $\mathsf{SC}^{=}$ to the trace-relating setting, states that whenever $\mathsf{W}\!\downarrow$ produces a trace t that violates a target safety property, then W violates the source *interpretation* of the property, i.e., its image through $\tilde{\sigma}$.[7] The following theorem defines $\mathsf{SC}^{\sim}$ and its two equivalent formulations.

Theorem 3.6 (Trinitarian view for Safety). For a trace relation $\sim \subseteq \mathsf{Trace_S} \times \mathsf{Trace_T}$ and its corresponding property mappings $\tilde{\sigma}$ and $\tilde{\tau}$, the following are equivalent:

$$\mathsf{SC}^{\sim} \equiv \forall \mathsf{W} \forall \mathsf{t} \forall \mathsf{m} \leq \mathsf{t}.\ \mathsf{W}\!\downarrow \rightsquigarrow \mathsf{t} \Rightarrow \exists \mathsf{t}' \geq \mathsf{m} \exists \mathsf{s} \sim \mathsf{t}'.\ \mathsf{W} \rightsquigarrow \mathsf{s}$$
$$\mathsf{SP}^{\tilde{\sigma}} \equiv \forall \mathsf{W}\ \forall \pi_{\mathsf{T}} \in \mathsf{Safety_T}.\ \mathsf{W} \models \tilde{\sigma}(\pi_{\mathsf{T}}) \Rightarrow \mathsf{W}\!\downarrow \models \pi_{\mathsf{T}}$$
$$\mathsf{TP}^{\mathit{Safe} \circ \tilde{\tau}} \equiv \forall \mathsf{W}\ \forall \pi_{\mathsf{S}} \in 2^{\mathsf{Trace_S}}.\ \mathsf{W} \models \pi_{\mathsf{S}} \Rightarrow \mathsf{W}\!\downarrow \models (\mathit{Safe} \circ \tilde{\tau})(\pi_{\mathsf{S}})$$

$$\begin{tikzcd}
 & \mathsf{SC}^{\sim} \arrow[dl, Leftrightarrow] \arrow[dr, Leftrightarrow] & \\
\mathsf{SP}^{\tilde{\sigma}} \arrow[rr, Leftrightarrow] & & \mathsf{TP}^{\mathit{Safe} \circ \tilde{\tau}}
\end{tikzcd}$$

Coherent with the informal meaning we aimed to give to $\mathsf{SC}^{\sim}$, $\mathsf{SP}^{\tilde{\sigma}}$ quantifies over target safety properties, while $\mathsf{TP}^{\mathit{Safe} \circ \tilde{\tau}}$ 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

[7] At least one other symmetric generalization is possible: For $\pi_{\mathsf{S}} \in \mathsf{Safety_S}$ defined by $M = \{m\}$, if $\mathsf{W}\!\downarrow$ produces a trace t that violates the target interpretation of π_{S}, i.e., $\tilde{\tau}(\pi_{\mathsf{S}})$, then W produces $\mathsf{s} \geq m$ thus violating π_{S}.

best over-approximates π_{T}.[8] More precisely, *Safe* is a closure operator on target properties, with $\mathsf{Safety}_{\mathsf{T}} = \left\{ Safe(\pi_{\mathsf{T}}) \;\middle|\; \pi_{\mathsf{T}} \in 2^{\mathsf{Trace}_{\mathsf{T}}} \right\}$ being the class of target safety properties.

In Figure 2 the blue and red ellipses represent source and target properties properties respectively and are connected by $\tilde{\tau} \leftrightarrows \tilde{\sigma}$. The red ellipse is the class of all target safety properties. $Safe \leftrightarrows id$ is a Galois connection between target properties and the target safety properties, as *Safe* is a closure operator [?]. Finally, the composition of Galois connections is still a Galois connection [?]. Hence,

$$Safe \circ \tilde{\tau} : 2^{\mathsf{Trace}_{\mathsf{S}}} \leftrightarrows \mathsf{Safety}_{\mathsf{T}} : \tilde{\sigma}$$

is a Galois connection between source properties and target safety properties, that we used to prove the equivalence $\mathsf{TP}^{Safe \circ \tilde{\tau}} \iff \mathsf{SP}^{\tilde{\sigma}}$ (🐓). We notice that this argument generalizes to arbitrary closure operators on target properties (🐓). We come back to this in §6, where more such results will be needed when considering other classes of properties being preserved by secure compilers. Now, we define the trinity for arbitrary hyperproperties, not just the subset-closed ones.

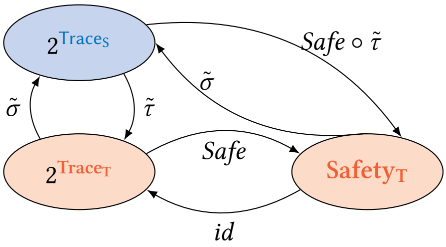

$2^{\mathsf{Trace}_{\mathsf{S}}}$
$Safe \circ \tilde{\tau}$
$\tilde{\sigma}$
$\tilde{\sigma}$
$\tilde{\tau}$
$Safe$
$2^{\mathsf{Trace}_{\mathsf{T}}}$
$\mathsf{Safety}_{\mathsf{T}}$
id

Fig. 2. Composition of $\tilde{\tau} \leftrightarrows \tilde{\sigma}$ and $Safe \leftrightarrows id$.

3.3 Preserving Non-Subset Closed Hyperproperties

Subset-closed hyperproperties are not expressive enough to all capture interesting properties, e.g., possibilistic notions of information-flow [?], so we aim to briefly discuss the preservation of *arbitrary* hyperproperties. In general, one cannot lift a Galois connection over trace properties to a Galois connection over arbitrary hyperproperties.
While two out of three of the criteria we introduce in this section are equivalent under no assumptions ($\mathsf{HC}^{\sim} \iff \mathsf{HP}^{\tilde{\tau}}$), for a comparison with the third one we require that no information is lost in the pre or post composition of τ and σ. For this, we label the trinity in Theorem 3.8 as *weak*.

To start, we note that the following strengthening of $\mathsf{CC}^{=}$, denoted $\mathsf{HC}^{=}$, is equivalent to the preservation of arbitrary hyperproperties. Here, $\mathsf{beh}(\mathsf{W})$ is the set of all traces of W. [CA: I don't have a reference for this, but it is folklore and immediate to proof. – If I have time I will add a paper proof in the appendix]

Theorem 3.7 ($\mathsf{HC}^{=}$, HP)**.** The followings are equivalent

$$\mathsf{HC}^{=} \equiv \forall \mathsf{W}.\ \mathbf{beh}(\mathsf{W}\downarrow) = \mathsf{beh}(\mathsf{W})$$

$$\mathsf{HP} \equiv \forall \mathsf{W}\ \forall H \in 2^{2^{\mathsf{Trace}}}.\ \mathsf{W} \models H \iff \mathsf{W}\downarrow \models H$$

[8] $Safe(\pi_{\mathsf{T}}) = \cap \{ \mathsf{S}_{\mathsf{T}} \mid \pi_{\mathsf{T}} \subseteq \mathsf{S}_{\mathsf{T}} \wedge \mathsf{S}_{\mathsf{T}} \in \mathsf{Safety}_{\mathsf{T}} \}$ is the topological closure in the topology where safety properties coincide with the closed sets (see, e.g., ?] and ?]).

$\mathsf{HC}^{=}$ requires that the behavior of $\mathsf{W}\!\downarrow$ is exactly the same as the behavior of W. We generalize this to the trace-relating setting, by requiring that the behavior of $\mathsf{W}\!\downarrow$ coincide with the target interpretation of the source properties describing the behavior of W.[9]

Theorem 3.8 (Weak Trinity for Hyperproperties 🐓). For a trace relation $\sim\, \subseteq \mathsf{Trace_S} \times \mathbf{Trace_T}$ and induced property mappings $\tilde\sigma$ and $\tilde\tau$, we have:
$\mathsf{HC}^{\sim} \iff \mathsf{HP}^{\tilde\tau}$;
if $\tilde\tau \leftrightarrows \tilde\sigma$ is a Galois insertion (i.e., $\tilde\tau \circ \tilde\sigma = id$), then $\mathsf{HC}^{\sim} \Rightarrow \mathsf{HP}^{\tilde\sigma}$,
if $\tilde\sigma \leftrightarrows \tilde\tau$ is a Galois reflection (i.e., $\tilde\sigma \circ \tilde\tau = id$), then $\mathsf{HP}^{\tilde\sigma} \Rightarrow \mathsf{HP}^{\tilde\tau}$,

$$\begin{aligned}
\mathsf{HC}^{\sim} &\equiv \forall \mathsf{W}.\ beh(\mathsf{W}\!\downarrow) = \tilde\tau(beh(\mathsf{W}))\\
\mathsf{HP}^{\tilde\tau} &\equiv \forall \mathsf{W}\ \forall \mathsf{H_S}.\ \mathsf{W} \models \mathsf{H_S} \Rightarrow \mathsf{W}\!\downarrow\, \boldsymbol{\models}\, \tilde\tau(\mathsf{H_S})\\
\mathsf{HP}^{\tilde\sigma} &\equiv \forall \mathsf{W}\ \forall \mathbf{H_T}.\ \mathsf{W} \models \tilde\sigma(\mathbf{H_T}) \Rightarrow \mathsf{W}\!\downarrow\, \boldsymbol{\models}\, \mathbf{H_T}
\end{aligned}$$

$$\begin{tikzcd}
& \mathsf{HC}^{\sim} \arrow[dl, Rightarrow, "\text{Insertion}"'] \arrow[dr, Leftrightarrow] & \\
\mathsf{HP}^{\tilde\sigma} \arrow[rr, Rightarrow, "\text{Reflection}"] & & \mathsf{HP}^{\tilde\tau}
\end{tikzcd}$$

In other words, it is still possible (and sound) to deduce a source obligation for a given target hyperproperty $\mathbf{H_T}$ ($\mathsf{HC}^{\sim} \Rightarrow \mathsf{HP}^{\tilde\sigma}$) when no information is lost in the composition $\tilde\tau \circ \tilde\sigma$. Dually, $\mathsf{HP}^{\tilde\tau}$ (and hence $\mathsf{HC}^{\sim}$) is a consequence of $\mathsf{HP}^{\tilde\sigma}$ when no information is lost in composing in the other direction, $\tilde\sigma \circ \tilde\tau$.

3.4 Comparing the Presented Criteria

At this point we have presented four trinities of criteria that preserve trace properties, subset-closed hyperproperties, safety properties and arbitrary hyperproperties. Figure 3 sums up our trinities and orders them according their relative strength.

$$\begin{tikzcd}
& \mathsf{HC}^{\sim} \arrow[ddl, Rightarrow, "\text{Insertion}"'] \arrow[ddr, Leftrightarrow] \arrow[drrrr, Rightarrow] & & & & & & & & & \\
& & & & & \mathsf{CC}^{\sim} \arrow[dl, Leftrightarrow] \arrow[dr, Leftrightarrow] \arrow[ddll, Leftrightarrow, bend right] \arrow[ddrr, Leftrightarrow, bend left] \arrow[drrrr, Rightarrow] & & & & & \\
\mathsf{HP}^{\tilde\sigma} \arrow[rr, Rightarrow, "\text{Reflection}"] & & \mathsf{HP}^{\tilde\tau} & & \mathsf{TP}^{\tilde\sigma} \arrow[rr, Leftrightarrow] & & \mathsf{TP}^{\tilde\tau} & & & \mathsf{SC}^{\sim} \arrow[ddl, Leftrightarrow] \arrow[ddr, Leftrightarrow] & \\
& & & \mathsf{SCHP}^{Cl_{\subseteq}\circ\tilde\sigma} \arrow[rrrr, Leftrightarrow] & & & & \mathsf{SCHP}^{Cl_{\subseteq}\circ\tilde\tau} & & & \\
& & & & & & & & \mathsf{SP}^{\tilde\sigma} \arrow[rr, Leftrightarrow] & & \mathsf{TP}^{Safe\circ\tilde\tau}
\end{tikzcd}$$

Fig. 3. Generalization of Compiler Correctness and its trace-relating variations.

[CA: I think this is necessary. Still no reference nor a proof but immediate.] In §6 we will also consider, in the setting of *secure* compilation, the class of safety hyperproperties or hypersafety, and relational hyperproperties. In the setting of *correct* compilation – that focuses only on whole programs – it is straightforward to show that the trinity for hypersafety coincides with the one for safety properties in the same way the trinity of trace properties and subset-closed hyperproperties coincide. Similarly the trinity for relational hyperproperties coincides with the one for hyperproperties.

4 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

[9] At least one generalization is possible: $\tilde\sigma(\mathbf{beh}(\mathsf{W}\!\downarrow)) = \mathsf{beh}(\mathsf{W})$. In this case, $\mathsf{HC}^{\sim} \iff \mathsf{HP}^{\tilde\sigma}$ holds unconditionally while the other two implications hold under the same, but swapped, hypotheses from Theorem 3.8.

target behavior (§4.1), target languages with resource exhaustion that cannot happen in the source (§4.2), changes in the representation of values (§4.3), and differences in the granularity of data and observable events (§4.4).

4.1 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 4.1 (CompCert-like Undefined Behavior Relation)**.** Source and target traces are sequences of events drawn from Σ, where $\mathit{Wrong} \in \Sigma$ is a terminal event that represents an undefined behavior. We then use the trace relation defined in the introduction:

$$\mathsf{s} \sim \mathsf{t} \ \equiv\ \mathsf{s} = \mathsf{t} \vee \exists m \leq \mathsf{t}.\ \mathsf{s} = m \cdot \mathit{Wrong}$$

Each trace of a target program produced by a $\mathrm{CC}^{\sim}$ compiler either also is a trace of the original source program or 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 $\mathrm{CC}^{\sim}$.

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

$$\tilde{\sigma}(\pi_{\mathsf{T}}) = \{\mathsf{s} \mid \mathsf{s} \in \pi_{\mathsf{T}} \wedge \mathsf{s} \neq m \cdot \mathit{Wrong}\} \cup \{m \cdot \mathit{Wrong} \mid \forall \mathsf{t}.\ m \leq \mathsf{t} \Rightarrow \mathsf{t} \in \pi_{\mathsf{T}}\}$$
$$\tilde{\tau}(\pi_{\mathsf{S}}) = \{\mathsf{t} \mid \mathsf{t} \in \pi_{\mathsf{S}}\} \cup \{\mathsf{t} \mid \exists m \leq \mathsf{t}.\ m \cdot \mathit{Wrong} \in \pi_{\mathsf{S}}\}$$

These two mappings explain what a $\mathrm{CC}^{\sim}$ 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 does not have any undefined behavior (second disjunct). Indeed, if it had an undefined behavior, its continuations could exhibit the unwanted event. ⊡

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, or controlling a device that is hidden from the source language. Formally, the source and target do not have the same events any more. Thus, we consider a source alphabet $\Sigma_{\mathsf{S}} = \Sigma \cup \{\mathsf{Wrong}\}$, and a target alphabet $\Sigma_{\mathsf{T}} = \Sigma \cup \Sigma'$. The trace relation is defined in the same way and we obtain the same property mappings as above, except that target traces now have more events (some of which may be dangerous), 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 [?]. 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 §7.1 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.

4.2 Resource Exhaustion

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

Example 4.2 (Resource Exhaustion)**.** We consider traces made of events drawn from Σ_S in the source, and $\Sigma_T = \Sigma_S \cup \{\mathsf{Resource\_Limit\_Hit}\}$ in the target. Recall the trace relation for resource exhaustion:

$$\mathsf{s} \sim \mathsf{t} \;\equiv\; \mathsf{s} = \mathsf{t} \vee \exists m \leq \mathsf{s}.\ \mathsf{t} = m \cdot \mathsf{Resource\_Limit\_Hit}$$

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 (🐓):

$$\tilde{\sigma}(\pi_T) = \{\mathsf{s} \mid \mathsf{s} \in \pi_T\} \cap \{\mathsf{s} \mid \forall m \leq \mathsf{s}.\ m \cdot \mathsf{Resource\_Limit\_Hit} \in \pi_T\}$$

$$\tilde{\tau}(\pi_S) = \pi_S \cup \{m \cdot \mathsf{Resource\_Limit\_Hit} \mid \exists \mathsf{s} \in \pi_S.\ m \leq \mathsf{s}\}$$

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 source-to-target 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 [?] 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. To reason about safety, one would use the criteria presented in §3.2

Since it focuses on traces and not just safety, the compiler correctness theorem in CakeML is an instance of $\mathrm{CC}^{\sim}$ 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 §4.3 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 $\mathrm{CC}^{\sim}$ instance for a variant of a compiler from a WHILE language to a simple stack machine by Xavier Leroy [?]. 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: if the source execution takes a step from a configuration to another configuration emitting some event (which can be a silent event), then there are two possibilities for a related target configuration: either (i) it can take some steps to another configuration related to the second source configuration and emit the same event (as in a standard simulation); or (ii) it can take some steps to an error state without emitting any events. The latter corresponds to the case of a resource exhaustion error: the target execution can terminate early, producing only a prefix of the source execution trace, as allowed by the relation. ⊡

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 $\sim_{\mathrm{UB}}$ and a relation $\sim_{\mathrm{RE}}$ 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: $s \sim t \equiv s \sim_{\mathrm{UB}} t \vee s \sim_{\mathrm{RE}} t$. A compiler that is $CC^{\sim_{\mathrm{UB}}}$ or $CC^{\sim_{\mathrm{RE}}}$ is trivially $CC^{\sim}$, though the converse is not true.

4.3 Different Source and Target Values

This section first presents the common language formalisation (§4.3.1) that the following (§4.3.2) and later instances (§4.4 and §7.1) build upon. This shared language formalisation does not contain a key language feature, namely the expressions that generate actions and thus labels. This is because each instance deals with specific ways to generate actions, so each instance will define its own extension to each of the languages defined below. Additionally, each instance will define its own compiler and the trace relation used to attain $CC^{\sim}$.

4.3.1 Shared Source and Target Language Formalisation. The source language is a pure, statically typed expression language whose expressions e include naturals, booleans, a boolean conditional and a conditional for expressions that reduce to 0, arithmetic and relational operations and sequencing.

$$e ::= n \mid b \mid \text{if } e \text{ then } e \text{ else } e \mid \text{ifz } e \text{ then } e \text{ else } e \mid e \text{ op } e \mid e; e'$$

$$op ::= + \mid \times \mid \leq \mid == \qquad ty ::= B \mid N$$

Types ty are either N (naturals) or B (booleans) and typing is standard.

$$\overset{\text{(Type-nat)}}{\dfrac{}{\vdash n : N}} \qquad \overset{\text{(Type-bool)}}{\dfrac{}{\vdash b : B}} \qquad \overset{\text{(Type-plus-times)}}{\dfrac{\vdash e_1 : N \qquad \vdash e_2 : N \qquad \cdot = + \text{ or } \times}{\vdash e_1 \cdot e_2 : N}} \qquad \overset{\text{(Type-le)}}{\dfrac{\vdash e_1 : N \qquad \vdash e_2 : N}{\vdash e_1 \leq e_2 : B}}$$

$$\overset{\text{(Type-ite)}}{\dfrac{\vdash e_1 : B \qquad \vdash e_2 : ty \qquad \vdash e_3 : ty}{\vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : ty}} \qquad \overset{\text{(Type-izte)}}{\dfrac{\vdash e_1 : N \qquad \vdash e_2 : ty \qquad \vdash e_3 : ty}{\vdash \text{ifz } e_1 \text{ then } e_2 \text{ else } e_3 : ty}}$$

The language semantics deal with actions i, lists of actions is and expression results r. A list of actions is is a list of individual actions i, which are instance-dependant and thus presented later; the same holds for source traces s.

$$r ::= n \mid b \qquad i, s ::= \text{instance-specific} \qquad is ::= i \cdot is \mid \varnothing$$

The source language has a standard big-step operational semantics ($e \rightsquigarrow \langle is, r\rangle$) which tells how an expression e generates a list of actions and a result $\langle is, r\rangle$.

$$\overset{\text{(Sem-nat)}}{\dfrac{}{n \rightsquigarrow \langle \varnothing, n\rangle}} \qquad \overset{\text{(Sem-bool)}}{\dfrac{}{b \rightsquigarrow \langle \varnothing, b\rangle}} \qquad \overset{\text{(Sem-op-nat)}}{\dfrac{e_1 \rightsquigarrow \langle is_1, n_1\rangle \qquad e_2 \rightsquigarrow \langle is_2, n_2\rangle \qquad op \in \{+, \times\}}{e_1 \text{ op } e_2 \rightsquigarrow \langle is_1 \cdot is_2, (n_1 \text{ op } n_2)\rangle}}$$

$$\overset{\text{(Sem-le)}}{\dfrac{e_1 \rightsquigarrow \langle is_1, n_1\rangle \qquad e_2 \rightsquigarrow \langle is_2, n_2\rangle}{e_1 \leq e_2 \rightsquigarrow \langle is_1 \cdot is_2, (n_1 \leq n_2)\rangle}} \qquad \overset{\text{(Sem-ite)}}{\dfrac{e \rightsquigarrow \langle is, b\rangle \qquad b ? i = 1 : i = 2 \qquad e_i \rightsquigarrow \langle is_i, r_i\rangle}{\text{if } e \text{ then } e_1 \text{ else } e_2 \rightsquigarrow \langle is \cdot is_i, r_i\rangle}}$$

$$\overset{\text{(Sem-izte)}}{\dfrac{e \rightsquigarrow \langle is, n\rangle \qquad n == 0 ? i = 1 : i = 2 \qquad e_i \rightsquigarrow \langle is_i, r_i\rangle}{\text{ifz } e \text{ then } e_1 \text{ else } e_2 \rightsquigarrow \langle is \cdot is_i, r_i\rangle}} \qquad \overset{\text{(Sem-seq)}}{\dfrac{e \rightsquigarrow \langle is, r\rangle \qquad e' \rightsquigarrow \langle is', r'\rangle}{e; e' \rightsquigarrow \langle is \cdot is', r'\rangle}}$$

The target language is analogous to the source one, except that it is untyped, it only has naturals $\mathbf{n}$ and its only conditional is $\mathbf{ifz\ e\ then\ e\ else\ e}$.

$$\mathbf{e} ::= \mathbf{n} \mid \mathbf{e\ op\ e} \mid \mathbf{ifz\ e\ then\ e\ else\ e} \mid \mathbf{e; e'} \qquad \mathbf{op} ::= + \mid \times \qquad \mathbf{r} ::= \mathbf{n}$$
$$\mathbf{i}, \mathbf{t} ::= \text{ instance-specific} \qquad \mathbf{is} ::= \mathbf{i} \cdot \mathbf{is} \mid \varnothing$$

The semantics of the target language is also given in big-step style; since its rules are a subset of the source rules, they are omitted. Since we only have naturals and all expressions operate on them, no error result is possible in the target.

4.3.2 Different Source and Target Values. In this instance, we extend the source language with expressions to perform booleans and natural inputs, while the target only has expressions to input naturals. To compile the $\leq$, the target is also extended with a conditional that checks if an expression is less than another.

$$\mathsf{e} ::= \cdots \mid \mathsf{in\text{-}b} \mid \mathsf{in\text{-}n} \qquad \mathsf{i} ::= \mathsf{n} \mid \mathsf{b} \qquad \mathsf{s} ::= \langle \mathsf{is}, \mathsf{r} \rangle$$
$$\mathbf{e} ::= \cdots \mid \mathbf{in\text{-}n} \mid \mathbf{if\ e \leq e\ then\ e\ else\ e} \qquad \mathbf{i} ::= \mathbf{n} \qquad \mathbf{t} ::= \langle \mathbf{is}, \mathbf{r} \rangle$$

Source actions are boolean b and natural inputs n and source traces s are lists of actions is together with a final result r. Target actions are just natural inputs $\mathbf{n}$.

The source extensions respect typing and thus well-typed programs never produce error (🐓). The semantics of the extensions adds elements to the traces.

$$\text{(Type-in-b)}\ \frac{}{\vdash \mathsf{in\text{-}b} : \mathsf{B}} \qquad \text{(Type-in-n)}\ \frac{}{\vdash \mathsf{in\text{-}n} : \mathsf{N}} \qquad \text{(Sem-in-nat)}\ \frac{}{\mathsf{in\text{-}n} \rightsquigarrow \langle \mathsf{n} \cdot \varnothing, \mathsf{n} \rangle} \qquad \text{(Sem-in-bool)}\ \frac{}{\mathsf{in\text{-}b} \rightsquigarrow \langle \mathsf{b} \cdot \varnothing, \mathsf{b} \rangle}$$

$$\text{(Sem-itele)}\ \frac{\mathbf{e_1} \rightsquigarrow \langle \mathbf{is_1}, \mathbf{n_1} \rangle \quad \mathbf{e_2} \rightsquigarrow \langle \mathbf{is_2}, \mathbf{n_2} \rangle \quad n_1 \leq n_2 ? i = 3 : i = 4 \quad \mathbf{e_i} \rightsquigarrow \langle \mathbf{is_i}, \mathbf{n_i} \rangle}{\mathbf{if\ e_1 \leq e_2\ then\ e_3\ else\ e_4} \rightsquigarrow \langle \mathbf{is_1} \cdot \mathbf{is_2} \cdot \mathbf{is_i}, \mathbf{n_i} \rangle}$$

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

$$\begin{aligned}
\mathsf{n}{\downarrow} &= \mathbf{n} & \mathsf{true}{\downarrow} &= \mathbf{1} & \mathsf{e_1 + e_2}{\downarrow} &= \mathsf{e_1}{\downarrow} \mathbf{+} \mathsf{e_2}{\downarrow} \\
\mathsf{in\text{-}n}{\downarrow} &= \mathbf{in\text{-}n} & \mathsf{false}{\downarrow} &= \mathbf{0} & \mathsf{e_1 \leq e_2}{\downarrow} &= \mathbf{if}\ \mathsf{e_1}{\downarrow} \mathbf{\leq} \mathsf{e_2}{\downarrow}\ \mathbf{then\ 1\ else\ 0} \\
\mathsf{in\text{-}b}{\downarrow} &= \mathbf{in\text{-}n} & \mathsf{e_1 \times e_2}{\downarrow} &= \mathsf{e_1}{\downarrow} \mathbf{\times} \mathsf{e_2}{\downarrow} & \mathsf{if\ e_1\ then\ e_2\ else\ e_3}{\downarrow} &= \mathbf{ifz}\ \mathsf{e_1}{\downarrow}\ \mathbf{then}\ \mathsf{e_3}{\downarrow}\ \mathbf{else}\ \mathsf{e_2}{\downarrow} \\
\mathsf{e; e'}{\downarrow} &= \mathsf{e}{\downarrow} \mathbf{;} \mathsf{e'}{\downarrow} & & & \mathsf{ifz\ e_1\ then\ e_2\ else\ e_3}{\downarrow} &= \mathbf{ifz}\ \mathsf{e_1}{\downarrow}\ \mathbf{then}\ \mathsf{e_2}{\downarrow}\ \mathbf{else}\ \mathsf{e_3}{\downarrow}
\end{aligned}$$

When compiling an *if-then-else* the *then* and *else* branches of the source are swapped in the target because of the compilation of booleans.

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).

$$\mathsf{n} \sim \mathbf{n} \qquad \mathsf{true} \sim \mathbf{n} \quad \text{if } \mathbf{n} > 0 \qquad \mathsf{false} \sim \mathbf{0}$$

$$\text{(Empty)}\ \frac{}{\varnothing \sim \boldsymbol{\varnothing}} \qquad \text{(Cons)}\ \frac{\mathsf{i} \sim \mathbf{i} \quad \mathsf{is} \sim \mathbf{is}}{\mathsf{i} \cdot \mathsf{is} \sim \mathbf{i} \cdot \mathbf{is}} \qquad \text{(Nat)}\ \frac{\mathsf{is} \sim \mathbf{is} \quad \mathsf{n} \sim \mathbf{n}}{\langle \mathsf{is}, \mathsf{n} \rangle \sim \langle \mathbf{is}, \mathbf{n} \rangle} \qquad \text{(Bool)}\ \frac{\mathsf{is} \sim \mathbf{is} \quad \mathsf{b} \sim \mathbf{n}}{\langle \mathsf{is}, \mathsf{b} \rangle \sim \langle \mathbf{is}, \mathbf{n} \rangle}$$

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 $\mathbf{0}$ be replaceable by *both* $\mathsf{0}$ and false.

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

Theorem 4.3 ($\cdot\!\downarrow$ is correct 🐓). $\cdot\!\downarrow$ is CC$^{\sim}$.

Simulations with different traces. In the settings where Trace_S = $\boldsymbol{\mathit{Trace}}_T$, it is customary to prove compiler correctness showing a forward simulation (i.e., a simulation between source and target transition system); then, using determinacy [**? ?**] of the target language and input totality [**? ?**] (receptiveness) of the source, this forward simulation is flipped into a backward simulation (a simulation between target and source transition system), as described by **? ?**]. This *"flipping"* is useful because forward simulations are often much easier to prove (by induction on the transitions of the source) than backward ones. For the proof of Theorem 4.3 we had to show a *backward* simulation as it was not possible to define a forward one and then flip it. Hereafter we show the reason lies in the shape of trace relation itself and disccus when is possible to generalize the flipping to the trace-relating setting.

We first give the main idea of the flipping proof, when the inputs are the same in the source and the target [**? ?**]. 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 W_T that simulates a source program W_S, W_T is able to perform an input iff so is W_S: otherwise, say for instance that W_S performs an output, by forward simulation W_T would also perform an output, which is impossible because of determinacy. By input totality of the source, W_S must be able to perform the exact same input as W_T; using forward simulation and determinacy, the resulting programs must be related.

$$\begin{tikzcd}
W_S \arrow[r, dash, "="] \arrow[d, dashed, "i_1"'] & W_S \arrow[r, dash, "\mathcal{R}"] \arrow[d, "i_2"'] \arrow[dl, phantom, "\Leftarrow \text{By input totality}"] \arrow[dr, phantom, "\Leftrightarrow \text{By contradiction, using forward simulation and determinacy}"] & W_T \arrow[d, "i_1"] \\
\exists W_{S1} \arrow[rr, dash, dashed, bend right, "\mathcal{R}"', "\text{By forward simulation and determinacy}"'] & \cdot & W_{T1}
\end{tikzcd}$$

The trace relation from §4.3.2 is not injective (both 0 and false are mapped to 0), therefore these arguments do not apply: not all possible inputs of target programs are accounted for in the forward simulation. In order to flip a forward simulation into a backward one it's necessary that, for any source program W_S and target program W_T related by the forward simulation $\mathcal{R}$, the following diagram is satisfied

$$\begin{tikzcd}
 & W_S \arrow[r, dash, "\mathcal{R}"] \arrow[dl, dashed, "\exists i_{S2}"'] \arrow[d, "i_{S1}"] & W_T \arrow[d, "i_{T1}"] \arrow[dr, "i_{T2}"] & \\
\exists W_{S2} \arrow[rrr, dash, dashed, bend right, "\mathcal{R}"'] & W_{S1} \arrow[r, dash, "\mathcal{R}"] & W_{T1} & W_{T2}
\end{tikzcd}$$

where $i_{S1} \sim i_{T1}$
$i_{S1} \sim i_{T2}$
$i_{S2} \sim i_{T2}$

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 n (and not just 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 $W_S \;\mathcal{R}\; W_T$ and an input i_{T2}, we show that there is i_{S1} and 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 i_{S2}. From this we obtain $CC^{\sim}$.

In fact we showed that the flippable hypothesis is also sufficient to flip a forward simulation into a backward one, even in the trace-relating setting, and proved it in a general (i.e., language independent) 'flipping theorem' (🐓). We have also shown that if the relation ~ defines a bijection between the inputs of the source and the target, then any forward simulation is flippable, hence reobtaining the usual proof technique [? ?] as a special case.

4.4 Abstraction Mismatches

We now consider how to relate traces where a single source action is compiled to multiple target ones. To illustrate this, we extend our source language to output (nested) pairs of arbitrary size, and our target language to send values that have a fixed size. Concretely, the source is analogous to the language of §4.3, except that it does not have inputs (nor booleans for simplicity) but it has pairs. Additionally, it has an expression send e which can emit a (nested) pair e of values in a single action. Given that e reduces to a pair, e.g., $\langle \mathsf{v1}, \langle \mathsf{v2}, \mathsf{v3}\rangle\rangle$, expression send $\langle \mathsf{v1}, \langle \mathsf{v2}, \mathsf{v3}\rangle\rangle$ emits action $\langle \mathsf{v1}, \langle \mathsf{v2}, \mathsf{v3}\rangle\rangle$. That expression is eventually compiled into a sequence of individual sends in the target language **send v1 ; send v2 ; send v3**, since in the target, **send e** sends the value that **e** reduces to, but the language cannot send pairs (although it has pair constructs).

The source and target languages are formally extended (resp. in the first and second lines below) with pairs and sending constructs as follows. For reasons that we explain when the compiler is presented, we extend the target language with a let-in construct and variables. Finally, source traces are sequences of sent values i (which include nested pairs) and target traces are only sequences of natural numbers.

$$\mathsf{e} ::= \cdots \mid \langle \mathsf{e}, \mathsf{e}\rangle \mid \mathsf{e.1} \mid \mathsf{e.2} \mid \mathsf{send\ e} \qquad \mathsf{ty} ::= \mathsf{N} \mid \mathsf{ty} \times \mathsf{ty} \qquad \mathsf{i} ::= \mathsf{n} \mid \langle \mathsf{i}, \mathsf{i}\rangle \qquad \mathsf{s} ::= \mathsf{is}$$

$$\mathbf{e} ::= \cdots \mid \langle \mathbf{e}, \mathbf{e}\rangle \mid \mathbf{e.1} \mid \mathbf{e.2} \mid \mathbf{let\ x = e\ in\ e} \mid \mathbf{x} \mid \mathbf{send\ e} \qquad \mathbf{i} ::= \mathbf{n} \qquad \mathbf{t} ::= \mathbf{is}$$

The source additions are well-typed and their semantics is unsurprising; the semantics relies on the usual capture-avoiding substitution $[\mathbf{r}/\mathbf{x}]$ of a result **r** for a variable **x**

$$\text{(Type-send)}\ \frac{\vdash \mathsf{e} : \tau \times \tau'}{\vdash \mathsf{send\ e}} \qquad \text{(Type-pair)}\ \frac{\vdash \mathsf{e} : \tau \qquad \vdash \mathsf{e}' : \tau'}{\vdash \langle \mathsf{e}, \mathsf{e}'\rangle : \tau \times \tau'} \qquad \text{(Type-p1)}\ \frac{\vdash \mathsf{e} : \tau \times \tau'}{\vdash \mathsf{e.1} : \tau} \qquad \text{(Type-p2)}\ \frac{\vdash \mathsf{e} : \tau \times \tau'}{\vdash \mathsf{e.2} : \tau'} \qquad \text{(Type-send)}\ \frac{\vdash \mathbf{e} : \mathbf{N}}{\vdash \mathbf{send\ e}}$$

$$\text{(Eval-P1)}\ \frac{\mathsf{e} \rightsquigarrow \langle \mathsf{is}, \langle \mathsf{r}_1, \mathsf{r}_2\rangle\rangle}{\mathsf{e.1} \rightsquigarrow \langle \mathsf{is}, \mathsf{r}_1\rangle} \qquad \text{(Eval-P2)}\ \frac{\mathsf{e} \rightsquigarrow \langle \mathsf{is}, \langle \mathsf{r}_1, \mathsf{r}_2\rangle\rangle}{\mathsf{e.1} \rightsquigarrow \langle \mathsf{is}, \mathsf{r}_2\rangle} \qquad \text{(Eval-Pair)}\ \frac{\mathsf{e} \rightsquigarrow \langle \mathsf{is}, \mathsf{r}\rangle \qquad \mathsf{e}' \rightsquigarrow \langle \mathsf{is}', \mathsf{r}'\rangle}{\langle \mathsf{e}, \mathsf{e}'\rangle \rightsquigarrow \langle \mathsf{is} \cdot \mathsf{is}', \langle \mathsf{r}, \mathsf{r}'\rangle\rangle}$$

$$\text{(Eval-Send)}\ \frac{\mathsf{e} \rightsquigarrow \langle \mathsf{is}, \mathsf{r}\rangle}{\mathsf{send\ e} \rightsquigarrow \langle \mathsf{is} \cdot \mathsf{r}, \mathsf{r}\rangle} \qquad \text{(Sem)}\ \frac{\mathsf{e} \rightsquigarrow \langle \mathsf{is}, \mathsf{r}\rangle}{\mathsf{e} \rightsquigarrow \mathsf{is}} \qquad \text{(Eval-letin)}\ \frac{\mathbf{e} \rightsquigarrow \langle \mathbf{is}, \mathbf{r}\rangle \qquad \mathbf{e}'[\mathbf{r}/\mathbf{x}] \rightsquigarrow \langle \mathbf{is}', \mathbf{r}'\rangle}{\mathbf{let\ x = e\ in\ e}' \rightsquigarrow \langle \mathbf{is} \cdot \mathbf{is}', \mathbf{r}'\rangle}$$

The compiler is defined inductively on the type derivation of a source expression $(\cdot\!\downarrow\, : \vdash \mathsf{e} : \tau \to \mathbf{e})$. 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. This is the reason we need the let-in construct in the target, since we run the pair once (as the argument of the let-in) and then we send all of its projection, to avoid duplicating side effects. Technically, since it is defined on the type derivations of terms, the compiler is defined inductively on type derivations (and not simply on terms). Thus, compiling $e; e'$ would look like the following (using D as a metavariable to range over derivations).

$$\left(\dfrac{\dfrac{D}{\vdash e} \qquad \dfrac{D'}{\vdash e'}}{\vdash e; e'}\right)\Bigg\downarrow = \left(\dfrac{D}{\vdash e}\right)\Big\downarrow ; \left(\dfrac{D'}{\vdash e'}\right)\Big\downarrow$$

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. To differentiate this from the compiler of Section 4.3.2, this compiler has parentheses over its input.

$$\begin{array}{rl@{\qquad}rl}
(\vdash n : N)\downarrow & = n & (\vdash e.1 : \tau)\downarrow & = (\vdash e : \tau \times \tau')\downarrow.1 \\
(\vdash e \oplus e' : N)\downarrow & = (\vdash e : N)\downarrow \oplus (e' : N)\downarrow & (\vdash e.2 : \tau')\downarrow & = (\vdash e : \tau \times \tau')\downarrow.2 \\
(\vdash \langle e, e' \rangle : \tau \times \tau')\downarrow & = \langle (\vdash e : \tau)\downarrow, (\vdash e' : \tau')\downarrow \rangle & & \\
\left(\vdash \begin{array}{l} \text{if } e \\ \text{then } e \text{ else } e' \end{array}\right)\Bigg\downarrow & = \begin{array}{l} \textbf{if } (\vdash e : N)\downarrow \\ \textbf{then } (\vdash e)\downarrow \textbf{ else } (\vdash e')\downarrow \end{array} & (\vdash \text{send } e)\downarrow & = \begin{array}{l} \textbf{let } x = (\vdash e : \tau \times \tau')\downarrow \\ \textbf{in } \text{gensend}\,(x, \tau \times \tau') \end{array} \\
\left(\vdash \begin{array}{l} \text{ifz } e \\ \text{then } e \text{ else } e' \end{array}\right)\Bigg\downarrow & = \begin{array}{l} \textbf{ifz } (\vdash e : N)\downarrow \\ \textbf{then } (\vdash e)\downarrow \textbf{ else } (\vdash e')\downarrow \end{array} & (\vdash e; e')\downarrow & = (\vdash e)\downarrow ; (\vdash e')\downarrow
\end{array}$$

$$\text{gensend}\,(x, \tau) = \begin{cases} \textbf{send } x & \text{if } \tau = N \\ \text{gensend}\,(x, \tau').1; \text{gensend}\,(x, \tau'').2 & \text{if } \tau = \tau' \times \tau'' \end{cases}$$

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 i and a target trace t (a list of numbers), inductively on the structure of i.

$$\overset{\text{(Trace-Rel-N-N)}}{\dfrac{n \sim^0 n \qquad n' \sim^0 n'}{\langle n, n' \rangle \sim n \cdot n'}} \qquad \overset{\text{(Trace-Rel-N-M)}}{\dfrac{n \sim^0 n \qquad i \sim t}{\langle n, i \rangle \sim n \cdot t}} \qquad \overset{\text{(Trace-Rel-M-N)}}{\dfrac{i \sim t \qquad n \sim^0 n}{\langle i, n \rangle \sim t \cdot n}} \qquad \overset{\text{(Trace-Rel-M-M)}}{\dfrac{i \sim t \qquad i' \sim t'}{\langle i, i' \rangle \sim t \cdot t'}}$$

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 4.5). 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). Before proving that the compiler is correct we need Lemma 4.4. Intuitively, that lemma tells us that the way we break down a source sent value r into multiple target sends is correct.

$$\overset{\text{(Trace-Rel-Single)}}{\dfrac{s \sim t \qquad i \sim t'}{s \cdot i \sim t \cdot t'}}$$

Lemma 4.4 (gensend $(\cdot, \cdot)$ works). if $\text{gensend}\,(x, \tau \times \tau')[(\vdash r : \tau \times \tau')\downarrow / x] \rightsquigarrow t$ then $r \sim t$ (since r is necessarily a sent value i, that can be related to t).

Theorem 4.5 ($(\cdot)\downarrow$ 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 $\sim$.[10]

- The source-to-target mapping "forgets" about the way pairs are nested, but is faithful w.r.t. the values $\mathsf{v_i}$ contained in a message. Notice that source safety properties are always mapped to target safety properties. For instance, if $\pi_S \in \mathsf{Safety_S}$ prescribes that some bad number is never sent, then $\tilde{\tau}(\pi_S)$ prescribes the same number is never sent in the target and $\tilde{\tau}(\pi_S) \in \mathsf{Safety_T}$. Of course if $\pi_S \in \mathsf{Safety_S}$ prescribes that a particular nested pairing like $\langle 4, \langle 6, \langle 5, 7\rangle\rangle\rangle$ never happens, then $\tilde{\tau}(\pi_S)$ is still a target safety property, but the trivial one, since $\tilde{\tau}(\pi_S) = \top \in \mathsf{Safety_T}$.

5 TRACE-RELATING COMPILATION AND NONINTERFERENCE PRESERVATION

We now study the relation between trace-relating compilation and noninterference preservation. As mentioned earlier (§3.1), in the particular case where 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) [?]. But in the scenario where target observations are strictly more informative than source observations, this is not the case. In fact, as we will show, the best guarantee one may expect from a correct trace-relating compiler ($CC^{\sim}$) in such a setting is a *weakening* (or *declassification*) of target noninterference that matches the noninterference property satisfied in the source. In certain scenarios, it turns out that the noninterference property of interest in the target comes "for free", while in others, it does not, and therefore establishing noninterference requires an additional proof effort beyond $CC^{\sim}$. To formalize this reasoning, this section applies the trinitarian view of trace-relating compilation to the general framework of abstract noninterference (ANI) [?], clarifying the kind of noninterference preservation that follows from a given trace relation and correct compilation.

We first define NI and explain the issue of preserving source NI via a $CC^{\sim}$ compiler (§5.1). We then introduce ANI, which allows characterizing various forms of noninterference (§5.2), and formulate a theory of ANI preservation via $CC^{\sim}$, both with respect to a *timing insensitive* declassification (§5.3) and in general (§5.4). We also study how to deal with cases such as undefined behavior in the target (§5.5). We then answer the dual question, i.e., which source NI should be satisfied to guarantee that compiled programs are noninterfering with respect to target observers (§5.6). Finally, we use this formal development to analyze recent work on correct compilers with interesting noninterference guarantees [? ?], clarifying whether these guarantees follow from correctness alone or not (§5.7).

5.1 Noninterference and Trace-Relating Compilation

Intuitively, noninterference (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 $t^{\bullet}$ respectively.[11] 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:

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

[10] Making $\sim$ injective is a matter of adding open and close parenthesis actions in target traces.

[11] The exact shape of inputs and outputs depends on the scenario. For instance, inputs can be initial memories and outputs trace semantics of programs as in [? , Section 7], while for interactive programs one may want to consider streams like ?]. We only require the sets of input and output projections to be disjoint. Further information, such as the ordering of events, is part of the attacker/observer model or the declassification of noninterference itself.

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

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 by considering a scenario where target traces extend source traces by exposing the execution time. While source noninterference $\mathsf{NI_S}$ requires that private inputs do not affect public outputs, $\mathbf{NI_T}$ additionally requires that the execution time is not affected by varying private inputs.

To model the scenario described, we represent target traces as pairs of a source trace and a natural number that denotes the time spent to produce the trace (using ω for infinite time units). Formally, if $\mathsf{Trace_S}$ denotes the set of source traces, then $\mathbf{Trace_T} = \mathsf{Trace_S} \times \mathbb{N}^\omega$ is the set of target traces, where $\mathbb{N}^\omega \triangleq \mathbb{N} \cup \{\omega\}$.
Notice that if two source traces $\mathsf{s}_1, \mathsf{s}_2$ are low-equivalent then $\{\mathsf{s}_1, \mathsf{s}_2\} \in \mathsf{NI_S}$ and $\{(\mathsf{s}_1, 42), (\mathsf{s}_1, 42)\} \in \mathbf{NI_T}$, but $\{(\mathsf{s}_1, 42), (\mathsf{s}_2, 43)\} \notin \mathbf{NI_T}$ and $\{(\mathsf{s}_1, 42), (\mathsf{s}_2, 42), (\mathsf{s}_1, 43), (\mathsf{s}_2, 43)\} \notin \mathbf{NI_T}$.
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:

$$\mathsf{s} \sim \mathbf{t} \equiv \exists \mathbf{n}.\ \mathbf{t} = (\mathsf{s}, \mathbf{n})$$

A compiler is CC$^\sim$ for this trace relation 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 3.3 says that if W satisfies $\mathsf{NI_S}$, then $\mathsf{W}\!\downarrow$ satisfies $Cl_\subseteq \circ \tilde{\tau}(\mathsf{NI_S})$. This hyperproperty is however strictly weaker than $\mathbf{NI_T}$, as it contains for example $\{(\mathsf{s}_1, 42), (\mathsf{s}_2, 42), (\mathsf{s}_1, 43), (\mathsf{s}_2, 43)\}$, and one cannot conclude that $\mathsf{W}\!\downarrow$ is noninterfering in the target. It is easy to check that

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

the first equality coming from $\tilde{\tau}(\pi_\mathsf{S}) = \pi_\mathsf{S} \times \mathbb{N}^\omega$, and the second from $\mathsf{NI_S}$ being subset-closed. As we will see, this hyperproperty *can* be characterized as a form of NI, which one might call *timing-insensitive noninterference*, i.e., 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* [?].

5.2 Abstract Noninterference

Abstract noninterference (ANI) [?] is a generalization of NI whose formulation relies on *abstractions* (in the sense of Abstract Interpretation [?]) 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 [?].[12] Formally:

$$ANI^\rho_\phi = \{\pi \mid \forall t_1 t_2 \in \pi.\ \phi(t_1^\circ) = \phi(t_2^\circ) \Rightarrow \rho(t_1^\bullet) = \rho(t_2^\bullet)\}$$

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.

[12] To be precise, the original formulation of ANI by ?] 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.

The operators ϕ and ρ are defined over sets of (input and output projections of) traces, explicitly $\phi : 2^{Trace^\circ} \to 2^{Trace^\circ}$ and $\rho : 2^{Trace^\bullet} \to 2^{Trace^\bullet}$. When we write $\phi(t)$ like 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 (i.e., $\forall\pi^\bullet.\ \pi^\bullet \subseteq \rho(\pi^\bullet)$) —on the poset that is the powerset of (input and output projections of) traces ordered by inclusion [?].

5.3 Trace-Relating Compilation and ANI for Timing

We can now reformulate our example with observable execution times in target traces in terms of ANI. We have $\mathsf{NI_S} = ANI^{\phi}_{\rho}$ with $\phi = \rho = [\cdot]_{low}$. In this case, the hyperproperty that a compiled program $\mathsf{W}\!\downarrow$ satisfies whenever W satisfies $\mathsf{NI_S}$ can be described as an instance of ANI:

$$Cl_{\subseteq} \circ \tilde{\tau}(\mathsf{NI_S}) = ANI^{\boldsymbol{\rho}}_{\boldsymbol{\phi}}$$
$$\text{for } \boldsymbol{\phi} = \phi \text{ and } \boldsymbol{\rho}(\boldsymbol{\pi}) = \{(\mathsf{s}, \mathbf{n}) \mid \exists(\mathsf{s}_1, \mathbf{n}_1) \in \boldsymbol{\pi}.\ [\mathsf{s}^\bullet]_{low} = [\mathsf{s}^\bullet_1]_{low}\}$$

The definition of $\boldsymbol{\phi}$ tells us that the trace relation does not affect the selection abstraction, i.e., declassification is unaffected. The definition of $\boldsymbol{\rho}$ characterizes an observer that cannot distinguish execution times for noninterfering traces (notice that $\mathbf{n}_1$ in the definition of $\boldsymbol{\rho}$ is discarded). For instance, $\boldsymbol{\rho}(\{(\mathsf{s}, \mathbf{n}_1)\}) = \boldsymbol{\rho}(\{(\mathsf{s}, \mathbf{n}_2)\})$, for any s, $\mathbf{n}_1$, $\mathbf{n}_2$. Therefore, in this setting, we know explicitly through $\boldsymbol{\rho}$ that a $\mathrm{CC}^\sim$ compiler degrades source noninterference to target *timing-insensitive* noninterference.

5.4 Trace-Relating Compilation and ANI in General

While the particular $\boldsymbol{\phi}$ and $\boldsymbol{\rho}$ 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 $\mathrm{CC}^\sim$ compiler?

We can now answer this question generally (Theorem 5.1): 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 $\sim \triangleq \langle \overset{\circ}{\sim}, \overset{\bullet}{\sim} \rangle$) such that $\mathsf{s} \sim \mathbf{t} \iff \mathsf{s}^\circ \overset{\circ}{\sim} \mathbf{t}^\circ \land \mathsf{s}^\bullet \overset{\bullet}{\sim} \mathbf{t}^\bullet$. The trace relation $\sim$ corresponds to a Galois connection between the sets of trace properties $\tilde{\tau} \leftrightarrows \tilde{\sigma}$ as described in §2.2. Similarly, the pair $\overset{\circ}{\sim}$ and $\overset{\bullet}{\sim}$ corresponds to a pair of Galois connections, $\tilde{\tau}^\circ \leftrightarrows \tilde{\sigma}^\circ$ and $\tilde{\tau}^\bullet \leftrightarrows \tilde{\sigma}^\bullet$, between the sets of input and output properties. In the timing example, time is an output so we have $\sim \triangleq \langle =, \overset{\bullet}{\sim} \rangle$ and $\overset{\bullet}{\sim}$ is defined as $\mathsf{s}^\bullet \overset{\bullet}{\sim} \mathbf{t}^\bullet \equiv \exists \mathbf{n}.\ \mathbf{t}^\bullet = (\mathsf{s}^\bullet, \mathbf{n})$.

Theorem 5.1 (Compiling ANI). Assume traces of source and target languages are related via $\sim \subseteq \mathsf{Trace_S} \times \mathbf{Trace_T}$, $\sim \triangleq \langle \overset{\circ}{\sim}, \overset{\bullet}{\sim} \rangle$ such that $\overset{\circ}{\sim}$ and $\overset{\bullet}{\sim}$ are both total maps from target to source traces, and $\overset{\circ}{\sim}$ is surjective. Assume $\downarrow$ is a $\mathrm{CC}^\sim$ compiler, and $\phi \in uco(2^{\mathsf{Trace}^\circ_\mathsf{S}})$, $\rho \in uco(2^{\mathsf{Trace}^\bullet_\mathsf{S}})$.
If W satisfies ANI^{ρ}_{ϕ}, then $\mathsf{W}\!\downarrow$ satisfies $ANI^{\boldsymbol{\rho}^\#}_{\boldsymbol{\phi}^\#}$, where $\boldsymbol{\phi}^\#$ and $\boldsymbol{\rho}^\#$ are defined as:

$$\boldsymbol{\phi}^\# = g^\circ \circ \phi \circ f^\circ \qquad\qquad \boldsymbol{\rho}^\# = g^\bullet \circ \rho \circ f^\bullet$$
$$f^\circ(\boldsymbol{\pi}^\circ) = \{\mathsf{s}^\circ \mid \exists \mathbf{t}^\circ \in \boldsymbol{\pi}^\circ.\ \mathsf{s}^\circ \overset{\circ}{\sim} \mathbf{t}^\circ\} \qquad\qquad g^\circ(\pi^\circ_\mathsf{S}) = \left\{\mathbf{t}^\circ \,\middle|\, \forall \mathsf{s}^\circ.\ \mathsf{s}^\circ \overset{\circ}{\sim} \mathbf{t}^\circ \Rightarrow \mathsf{s}^\circ \in \pi^\circ_\mathsf{S}\right\}$$

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

Moreover, we can prove that if $\stackrel{\bullet}{\sim}$ is surjective, then $ANI^{\rho^\#}_{\phi^\#} \subseteq Cl_{\subseteq} \circ \tilde{\tau}(ANI^{\rho}_{\phi})$. Therefore, the derived guarantee $ANI^{\rho^\#}_{\phi^\#}$ is at least as strong as the hyperproperty (a priori different from some noninterference) that follows by just knowing that the compiler ↓ is $CC^{\sim}$.

The target abstract noninterference has to be intended as the *best correct approximation* of the source one. The mappings $f^\circ \leftrightarrows g^\circ$ are the existential and universal images of the relation $\stackrel{\circ}{\sim}_{swap} \subseteq \mathsf{Trace_T} \times \mathsf{Trace_S}$, defined by $\mathsf{t}^\circ \stackrel{\circ}{\sim}_{swap} \mathsf{s}^\circ$ if and only if $\mathsf{s}^\circ \stackrel{\circ}{\sim} \mathsf{t}^\circ$. Therefore f° and g° are lower and upper adjoints, respectively (§2). The operator $\phi^\#$ is the best correct approximation of ϕ w.r.t to $f^\circ \leftrightarrows g^\circ$ [?] (hence the choice of the $(\_)^\#$ notation). A similar result holds for $\rho^\#$.

Coming back to our example above, we can formally recover the intuitively-justified definitions, i.e., $\phi^\# = g^\circ \circ \phi \circ f^\circ = \phi$ and $\rho^\# = g^\bullet \circ \rho \circ f^\bullet = \rho$.

5.5 Noninterference and Undefined Behavior

As stated above, Theorem 5.1 does not apply to several scenarios from §4 such as undefined behavior (§4.1). Indeed, in these cases, the relation $\stackrel{\bullet}{\sim}$ is not a total map. Nevertheless, we can still exploit our framework to reason about the impact of compilation on noninterference.

Let us consider $\sim \triangleq \langle \stackrel{\circ}{\sim}, \stackrel{\bullet}{\sim} \rangle$ where $\stackrel{\circ}{\sim}$ is any total and surjective map from target to source inputs (e.g., equality) and $\stackrel{\bullet}{\sim}$ is defined as $\mathsf{s}^\bullet \stackrel{\bullet}{\sim} \mathsf{t}^\bullet \equiv \mathsf{s}^\bullet = \mathsf{t}^\bullet \vee \exists m^\bullet \leq \mathsf{t}^\bullet .\ \mathsf{s}^\bullet = m^\bullet \cdot \mathit{Wrong}$. Intuitively, a $CC^{\sim}$ compiler guarantees noninterference for the compiled program, provided that the target attacker cannot exploit undefined behavior to learn private information. This intuition can be made formal by the following theorem.

Theorem 5.2 (Relaxed Compiling ANI). Relax the assumptions of Theorem 5.1 by allowing $\stackrel{\bullet}{\sim}$ to be *any* output trace relation. If W satisfies ANI^{ρ}_{ϕ}, then $\mathsf{W}\!\downarrow$ satisfies $ANI^{\rho^\#}_{\phi^\#}$ where $\phi^\#$ is defined as in Theorem 5.1, and $\rho^\#$ is such that:

$$\forall \mathsf{s}\ \mathsf{t}.\ \mathsf{s}^\bullet \stackrel{\bullet}{\sim} \mathsf{t}^\bullet \Rightarrow \ \rho^\#(\mathsf{t}^\bullet) = \rho^\#(\tilde{\tau}^\bullet(\rho(\mathsf{s}^\bullet))) \tag{1}$$

Technically, instead of giving us a *definition* of $\rho^\#$, the theorem gives a *property* of it. The property states that, given a target output trace $\mathsf{t}^\bullet$, the attacker cannot distinguish it from any other target output traces produced by other possible compilations ($\tilde{\tau}^\bullet$) of the source trace s it relates to, up to the observational power of the source level attacker ρ. Therefore, given a source attacker ρ, the theorem characterizes a *family* of attackers that cannot observe any interference for a correctly compiled noninterfering program. Notice that the target attacker $\rho^\top \triangleq \lambda\_.\ \top$ satisfies the premise of the theorem, but defines a trivial hyperproperty, so that we cannot prove in general that $ANI^{\rho^\#}_{\phi^\#} \subseteq Cl_{\subseteq} \circ \tilde{\tau}(ANI^{\rho}_{\phi})$. Also, this degenerate attacker $\rho^\top$ shows that the family of attackers described in Theorem 5.2 is nonempty, which ensures the existence of a most powerful attacker among them [?].

5.6 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 [? ?].

Let us consider the languages from §4.4 extended with the ability to accept inputs as (pairs of) values. It is easy to show that the compiler described in §4.4 (extended to treat the new input expressions homomorphically) is still $CC^{\sim}$: given a target trace t with the same inputs of the source one (i.e., $s^{\circ} = t^{\circ}$), the compiler of §4.4 ensures that t simulates the same outputs of s (i.e., $s^{\bullet} \mathrel{\dot{\sim}} t^{\bullet}$). Assume that we want to satisfy a given notion of target noninterference after compilation, i.e., $W{\downarrow}\models ANI^{\rho}_{\phi}$. Recall that the observational power of the target attacker, ρ, 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 $\mathring{\sim}$ is not the identity), this argument is valid only when ϕ can be represented in the source. More precisely, ϕ must consider as equivalent all target inputs that are related to the same source input, because in the source it is not possible to have a finer distinction of inputs. This intuitive correspondence can be formalized as follows.

Theorem 5.3 (Target ANI by source ANI). Let $\phi \in uco(2^{\mathrm{Trace}^{\circ}_{\mathrm{T}}})$, $\rho \in uco(2^{\mathrm{Trace}^{\bullet}_{\mathrm{T}}})$ and $\dot{\sim}$ a total and surjective map from source outputs to target ones and assume that

$$\forall s\ t.\ s^{\circ} \mathrel{\mathring{\sim}} t^{\circ} \Rightarrow \phi(t^{\circ}) = \phi(\tilde{\tau}^{\circ}(s^{\circ}))$$

If $\cdot{\downarrow}$ is a $CC^{\sim}$ compiler and W satisfies $ANI^{\rho^{\#}}_{\phi^{\#}}$, then $W{\downarrow}$ satisfies ANI^{ρ}_{ϕ} for

$$\phi^{\#} = \tilde{\sigma}^{\circ} \circ \phi \circ \tilde{\tau}^{\circ} \qquad\qquad \rho^{\#} = \tilde{\sigma}^{\bullet} \circ \rho \circ \tilde{\tau}^{\bullet}$$

5.7 Analyzing Noninterference Preserving Compilers

The results presented in this section formalize and generalize some intuitive facts about compiler correctness and noninterference, clarifying which noninterference property follows "for free" from trace-relating compiler correctness. Of course, in the general case, compiler correctness alone is not a strong enough criterion for dealing with many security properties [? ?]. This section exploits our ANI-based framework and results to analyze two compilers from the recent literature [? ?] that are both proven to be correct and to preserve two interesting notions of noninterference: cryptographic constant time (§5.7.1) and value-dependent noninterference (§5.7.2). For each, we explain how to express compiler correctness as an instance of $CC^{\sim}$, describe the noninterference property that is implied by the trace relation and the correctness result, and compare it with the noninterference properties of interest as established by their authors.

5.7.1 *A Correct Compiler Preserving Cryptographic Constant Time.*

?] provide a correct compiler (as an extension of CompCert) that also preserves cryptographic constant time (CT). CT is a security property stating that the runtime of a program does not depend on its secret, and thus an attacker cannot extrude secrets of a program by observing its execution time. A CT-preserving compiler takes code that is CT and generates code that also is CT. Thus, a CT-preserving compiler must translate runtime-equivalent source programs into runtime-equivalent target ones. Notice that it is not necessary for the leakage of target programs to be the same of their source counterparts, rather: source programs with the same leakage must be compiled to target programs with the same leakage.

?] prove CT preservation for seventeen passes of CompCert. The authors partition the seventeen steps in four categories depending on the proof technique they use to show CT preservation. Every category proves an instance of $CC^{\sim}$ by improving on the existing CompCert simulation. In three out of the four cases this is sufficient to also prove CT preservation, while for the last category a further proof is necessary. In what follows, we first encode CT as an instance of abstract noninterference, i.e., show for which operators $CT = ANI^{\rho_{CT}}_{\phi_{CT}}$ and then use our framework to understand why modifying CompCert simulation is sufficient in the first three categories but not in the last one. For each category Theorem 5.2 applies, so that no ρ that respects Equation 1 can notice any interference on compiled programs that were source constant-time. In the first three categories the attacker that defines CT – ρ_{CT} – respects the equation [13] i.e.,

$$\forall \mathsf{s}^{\bullet} \mathsf{t}^{\bullet}.\ \mathsf{s}^{\bullet} \mathrel{\dot{\sim}} \mathsf{t}^{\bullet} \Rightarrow \rho_{CT}(\mathsf{t}^{\bullet}) = \rho_{CT}(\tilde{\tau}^{\bullet}(\rho_{CT}(\mathsf{s}^{\bullet}))) \tag{2}$$

and CT preservation is therefore a consequence of $CC^{\sim}$. In the last category ρ_{CT} does not respect Equation 2 and the authors have to prove an additional theorem, the *CT-diagram*.

Trace Model and CT as an instance of ANI. The formal definition of CT is given by extending the semantics of the languages in CompCert and enriching the traces of input and output events with leakages. Leakages are results of execution steps that involve conditional branching or memory access. A program is CT w.r.t. a certain relation over program states φ [**?** , Definition 3.2] iff for every two initial states i, i' such that $\varphi(i, i')$, the leakages that can be observed are the same. Notice that in [**?** , Definition 3.2] the secret is stored in the program states and defined by φ, therefore in order to regard CT as an instance of abstract noninterference program states will be regarded as inputs and events together with their leakages as outputs. More precisely a trace t is a sequence of of triples (i, e, j) where i and j are program states and e an event in the instrumented semantics, i.e., input/output event and associated leakage.

We consider:

- ϕ_{CT} to be (the uco corresponding to) the relation defined by $t_1^{\circ}\ \phi_{CT}\ t_2^{\circ}$ iff t_1°, t_2° have the same length with $t_1^{\circ} = (i_0, i_1), (i_1, i_2), \ldots, t_2^{\circ} = (j_0, j_1), (j_1, j_2), \ldots$ and $\forall n.\ \varphi(i_n, j_n)$.

- ρ_{CT} to be (the uco corresponding to) the relation defined by $t_1^{\bullet}\ \rho_{CT}\ t_2^{\bullet}$ iff $t_1^{\bullet}, t_2^{\bullet}$ have the same length with $t_1^{\bullet} = e_0, e_1, \ldots, t_2^{\bullet} = f_0, f_1, \ldots$ and $\forall n.\ \mathit{leak}(e_n) = \mathit{leak}(f_n)$, where $\mathit{leak}(e)$ denotes the leakage in the event e (projection of e on the leak-only semantics [**?**]).

It is easy to check that $CT = ANI^{\rho_{CT}}_{\phi_{CT}}$ for the ϕ_{CT} and ρ_{CT} given above.

We now present more details for each of the four proof techniques adopted by **?**]. Since CT is defined only for *safe* programs [**?** , Definition 3.1] we can assume no undefined behavior is ever encountered and have a simpler presentation. We also omit $\phi^{\#}$ coming from the application of Theorem 5.2, as it always coincides with ϕ_{CT}. [CA: double check this fact]

Constant-time security preservation by leakage preservation (? , Section 5.2]). For compilation passes that belong to this category, the authors prove that the source leakage is preserved exactly in the target. Thus in this simple case, the theorem proved is $CC^{\sim}$ where $\dot{\sim}$ is point-wise equality of events together with leakages, $\tilde{\tau}^{\bullet}$ the identity and ρ_{CT} satisfies Equation 2 by idempotency of ρ_{CT},

$$\begin{aligned}
&\rho_{CT}(\tilde{\tau}^{\bullet}(\rho_{CT}(\mathsf{s}^{\bullet})) = && [\mathsf{s}^{\bullet} \mathrel{\dot{\sim}} \mathsf{t}^{\bullet} \Rightarrow \mathsf{s}^{\bullet} = \mathsf{t}^{\bullet}]\\
&\rho_{CT}(\tilde{\tau}^{\bullet}(\rho_{CT}(\mathsf{t}^{\bullet})) = && [\tilde{\tau}^{\bullet} = \lambda\, x.x]\\
&\rho_{CT}(\rho_{CT}(\mathsf{t}^{\bullet})) = && [\rho_{CT}\ \mathtt{idempotent}]\\
&\rho_{CT}(\mathsf{t}^{\bullet})
\end{aligned}$$

CT preservation from leakage-erasing simulation (? , Section 5.3]). In this case, $CC^{\sim}$ is proved for a relation that erases source leakage-only events, i.e., those events that do not contain inputs or outputs, but only the amount of leakage revealed. More precisely (see also [**?** , Fig. 8]) for $\mathsf{s}^{\bullet} = \mathsf{e}_0, \mathsf{e}_1, \ldots$ and $\mathsf{t}^{\bullet} = \mathsf{e}_0, \mathsf{e}_1, \ldots$ of the same length, $\mathsf{s}^{\bullet} \mathrel{\dot{\sim}} \mathsf{t}^{\bullet}$ iff

$$\forall k, \mathsf{e}_\mathsf{k} = \mathsf{e}_\mathsf{k} \vee (\mathsf{e}_\mathsf{k} = \epsilon \wedge \mathsf{e}_\mathsf{k}\ \textsf{is leak only})$$

[13] In each compilation step source and target traces are drawn from the same set so that ρ_{CT} can be applied to both source and target traces.

The property mapping associated to the above relation, $\tilde{\tau}^{\bullet}$, erases all leak-only events from the traces of a source property. If an attacker cannot notice at any point any difference in the leakages of two traces and we erase the leak-only events from them, the attacker will still not notice any difference on leakages, therefore it is easy to check that Equation 2 holds also in this case.

CT preservation via memory injection (? , Section 5.4]). This case is analogous to the one above, save that it rests on a more complex relation $\overset{\bullet}{\sim}$ involving a *memory injection* relation (see **?** , Definition 5.8]). [CA: Roughly speaking the first disjunct of the above $\overset{\bullet}{\sim}$ would use a map ι that is indeed complex as it needs also information about the states (i.e., inputs)] Intuitively $\overset{\bullet}{\sim}$ relates source and target traces that differ at most in leakages due to memory accesses. While in the previous case, leakages where simply erased, here they are modified and crucially with some uniformity. Reasoning as in the previous case, if an attacker cannot notice a difference in the leakages of two traces and we modify equal leakages of the same factor, the attacker will still not notice any difference on leakages, thus Equation 2 holds.

CT preservation from CT-diagram (? , Section 5.5]). In this case ρ_{CT} does not satisfy Equation 2 because the *counting simulation* ([**?** , Definition 5.10]) does not necessarily relate source and target leakages but only the inputs and outputs[14]. $\mathrm{CC}^{\sim}$ alone does not ensure that an attacker cannot observe any interference in the target leakages, in order to show preservation of CT the authors need to prove an extra condition, the so-called CT diagram [**?**].

5.7.2 *Value-dependent noninterference.*

?] introduce a compiler that provably preserves value-dependent noninterference (VDNI) for a concurrent language with shared variables. *Value-dependent* means that the secrecy level of a variable – *low* or *high* – may depend on the value of some other variable, called the control variable of the first, and therefore could change throughout its lifetime.

Preservation of VDNI for concurrent programs enjoys *compositionality*, meaning that it follows from the preservation of VDNI for each single thread [**?**] under certain conditions. As the compositionality result is orthogonal to our framework, we can study either (1) the preservation of VDNI for one local thread or for (2) the whole-program,

In the remainder of this section we focus on the preservation of VDNI for a single thread, that is proven by showing a *secure refinement* relation between source and compiled threads. Similarly to the previous section, the secure refinement is expressed via a cube diagram ([**?**], Figure 1), and can be proven directly [**?**] or split into more obligations [**?**].

As **?**] use a state transition based semantics, we first show how to encode this semantics into a trace model by defining the $\sim$ relation based on the secure refinement relation. We then show how to encode VDNI as an instance of abstract noninterference(i.e., both $\mathsf{VDNI_S} = \mathit{ANI}^{\rho}_{\phi}$ and $\mathbf{VDNI_T} = \mathit{ANI}^{\rho}_{\phi}$). Finally we apply Theorem 5.2 and conclude that if W satisfies $\mathsf{VDNI_S}$ then $\mathsf{W}\downarrow$ satisfies $\mathbf{VDNI_T}$ given that the trace relation $\sim$ has properties defined in [**?** , Theorem 5.1].

Source (`WHILE`) and target (`RISC`-like assembly) languages are equipped with a determined evaluation step semantics (i.e., a semantics where the only source of nondeterminism are external inputs, [**?**], Section 2) between thread-local configurations, which are triples of the form $\langle tps, mds, mem\rangle$. In such a configuration, *mds* is the access mode state for program variables and *mem* is a map relating global program variables to their values. Both of these components are common to the source and target language. The *tps* component denotes the thread-private state. In the source language, it is the program to be executed. In the target language, *tps* consists of the target program (labelled assembly-language instructions), of a program counter and of the set of thread-local registers. We

[14] The interested reader will notice the difference from the previous category by comparing condition (1) of Definition 5.10 and condition (1) of Definition 5.8 by **?**].

denote `WHILE` configurations by tuples of the form: $\langle \mathsf{tps}, mds, mem \rangle$ and `RISC` configurations by tuples of the form: $\langle \mathsf{tps}, mds, mem \rangle$.

Trace Model and Trace relation. We consider traces that are (possibly infinite) sequences of configurations. The traces produced by a program are the sequences of local configurations that the program may encounter during execution, according to the evaluation semantics. Let $\mathsf{s} = \langle \mathsf{tps}_1, mds_1, mem_1 \rangle, \langle \mathsf{tps}_2, mds_2, mem_2 \rangle \ldots$ be a source trace. The input projection is defined by $\mathsf{s}^\circ = \langle mds_1, mem_1 \rangle$ (the tuple consisting of the access modes and the memory in the first state) and the output projection is defined by $\mathsf{s}^\bullet = \mathsf{s}$ (the trace itself). Input/output projections are defined similarly for target traces.

We take the trace relation $\sim \subseteq \mathsf{Traces}_\mathsf{S} \times \boldsymbol{Trace}_\mathsf{T}$ to be the point-wise lifting of a secure refinement relation $\mathcal{R}$ ([?], Definition 6). Source and target configurations $\langle \mathsf{tps}, mds, mem \rangle \,\mathcal{R}\, \langle \mathsf{tps}, mds', mem' \rangle$ that are related coincide on the access mode and memory part (i.e., $mds = mds'$ and $mem = mem'$, ([?], Definition 4), so that $\overset{\circ}{\sim}$ is simply the identity and $\overset{\bullet}{\sim}$ coincides with $\sim$.

VDNI as abstract noninterference. A program satisfies VDNI ([?], Definition 2) if any two of its executions starting in low equivalent memories are related via a *strong low bisimulation modulo modes* (strong low bisimulation mm). Intuitively, a strong low bisimulation mm is a bisimulation that preserves low-equivalence. Preservation of VDNI is proved by ?] by showing that for every strong low-bisimulation mm $\mathcal{B}$ for source threads, there exists a target strong low bisimulation mm $\mathcal{B}$ such that if two source threads are related by $\mathcal{B}$, then the compiled threads are related by $\mathcal{B}$ ([?], Theorem 5.1).
The intuition for the encoding of VDNI as an instance of abstract noninterference is to model low equivalence through the operator ϕ, and bisimilarity through ρ. More rigorously, $\mathsf{VDNI}_\mathsf{S} = ANI^{\rho}_{\phi}$, where ϕ and ρ are defined as following.
For $\mathsf{s}^\circ = \langle mds_1, mem_1 \rangle$,

$$\phi(\mathsf{s}^\circ) = \left\{ \langle mds_1, mem'_1 \rangle \;\middle|\; mem_1 =^{Low}_{mds_1} mem'_1 \right\},$$

where $=^{Low}_{mds}$ is the low-equivalence modulo mds ([?], Definition 1).
For $\mathsf{s}^\bullet = \langle \mathsf{tps}_1, mds_1, mem_1 \rangle, \langle \mathsf{tps}_2, mds_2, mem_2 \rangle, \ldots,$

$$\begin{aligned}\rho(\mathsf{s}^\bullet) = \{ \langle \mathsf{tps}'_1, mds'_1, mem'_1 \rangle, \langle \mathsf{tps}'_2, mds'_2, mem'_2 \rangle, \ldots \mid \\ \forall i. \exists \mathcal{B}_\mathsf{i}.\ (\langle \mathsf{tps}_\mathsf{i}, mds_i, mem_i \rangle, \langle \mathsf{tps}'_\mathsf{i}, mds'_i, mem'_i \rangle) \in \mathcal{B}_\mathsf{i} \}\end{aligned}$$

where $\mathcal{B}_i$ denotes a strong low bisimulation modulo modes. Similarly $\mathbf{VDNI_T} = ANI^{\rho}_{\phi}$ where

$$\phi(\mathsf{t}^\circ) = \left\{ \langle mds_1, mem'_1 \rangle \;\middle|\; mem_1 =^{Low}_{mds_1} mem'_1 \right\},$$

$$\begin{aligned}\rho(\mathsf{t}^\bullet) = \{ \langle \mathsf{tps}'_1, mds'_1, mem'_1 \rangle, \langle \mathsf{tps}'_2, mds'_2, mem'_2 \rangle, \ldots \mid \\ \forall i. \exists \mathcal{B}_\mathsf{i}.\ (\langle \mathsf{tps}_\mathsf{i}, mds_i, mem_i \rangle, \langle \mathsf{tps}'_\mathsf{i}, mds'_i, mem'_i \rangle) \in \mathcal{B}_\mathsf{i} \}\end{aligned}$$

The relation $\mathcal{R}$ is a simulation, and therefore $CC^{\sim}$ holds. In order to apply Theorem 5.2 and conclude that whenever a source program W satisfies $\mathsf{VDNI}_\mathsf{S} = ANI^{\rho}_{\phi}$, then $\mathsf{W}{\downarrow}$ satisfies $\mathbf{VDNI_T} = ANI^{\rho}_{\phi}$, it is sufficient for ρ to satisfy Equation 1, that is

$$\boldsymbol{\rho}(\mathsf{t}^\bullet) = \boldsymbol{\rho}(\tilde{\tau}^\bullet(\rho(\mathsf{s}^\bullet)))$$

for $\mathsf{s}^\bullet \overset{\bullet}{\sim} \mathsf{t}^\bullet$. If one is willing to unfold all definitions, this amounts to show the set of traces “bismilar” to $\mathsf{t}^\bullet$ coincides with the set of traces that are bisimilar to some $\mathsf{t}'^\bullet$ and $\mathsf{s}'^\bullet \overset{\bullet}{\sim} \mathsf{t}'^\bullet$ for some $\mathsf{s}'^\bullet$ bisimilar to $\mathsf{s}^\bullet$. The “$\subseteq$” is immediate, while for the other one has to prove some properties of $\mathcal{R}$, the ones in the definition of `secure − refinement` (? , inlined above Theorem 5.1]) which entails preservation of low-equivalence as shown in ? , Theorem 5.1].

[CA: (not relevant for the moment) I am starting to conjecture that the equation from our theorem 5.2 describes what is the cube diagram to prove if one wants to preserve some noninterference. It generalizes the cube diagram when ANI is not expressed via equiv relations but via uco]

In summary, our framework makes it possible to precisely characterize the target noninterference properties that are implied by (trace-relating) correct compilation of source noninterfering programs. As we have shown, such properties are not necessarily as strong as desired. Crucially, the target noninterference property one gets *for free* for a given trace-relating correct compiler is a function of the trace relation under consideration. By considering more sophisticated trace relations, [CA: e.g., the one in the VDNI example] one could be able to get more interesting noninterference properties in the target *for free* —but this would likely come at the expense of a more challenging trace-relating compiler correctness proof.

6 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 [?]) or well-behaved target contexts that behave like source ones (as in compositional compiler correctness [? ? ? ?]).

Summary of the work of ?]. To account for this scenario, ?] 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 $CC^{=}$. 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 $\mathsf{C_S}$ satisfies π, then the compiled program $\mathsf{P}{\downarrow}$ plugged into any target context $\mathsf{C_T}$ 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:

$$\mathsf{RTP} \equiv \forall \mathsf{P}.\ \forall \pi.\ (\forall \mathsf{C_S}.\ \forall t. \mathsf{C_S}\,[\mathsf{P}] \rightsquigarrow t \Rightarrow t \in \pi) \Rightarrow (\forall \mathsf{C_T}.\ \forall \mathsf{t}.\ \mathsf{C_T}\,[\mathsf{P}{\downarrow}] \rightsquigarrow t \Rightarrow t \in \pi)$$

$$\mathsf{RTC} \equiv \forall \mathsf{P}.\ \forall \mathsf{C_T}. \forall t. \mathsf{C_T}\,[\mathsf{P}{\downarrow}] \rightsquigarrow t \Rightarrow \exists \mathsf{C_S}.\ \mathsf{C_S}\,[\mathsf{P}] \rightsquigarrow t$$

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 (C) it is linked with. Formally, $P \models_R \pi \overset{\text{def}}{=} \forall C, C\,[P] \models \pi$, where $\models$ is the same as before. Thus, we write more compactly:

$$\mathsf{RTP} \equiv \forall \pi.\ \forall \mathsf{P}.\ \mathsf{P} \models_{\mathsf{R}} \pi \Rightarrow \mathsf{P}{\downarrow} \models_{\mathsf{R}} \pi$$

All the criteria of ?] 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. 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.

Each category of criteria provides different kinds of security guarantees (confidentiality or integrity) for the code and data segments of programs. 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 may even regard code confidentiality. Naturally, these stronger guarantees are increasingly harder to enforce and prove.

All the criteria of **?**] are stated in a setting where source and target traces are the same. In this section, we extend their results to the trace-relating setting, obtaining trintarian views for secure compilation. There are many similarities with §2 which show up in the secure compilation setting too, but also some crucial differences. As in §2, the application of $\tilde{\sigma}$ or $\tilde{\tau}$, may lose the information that a property belongs to the class *Safety*, or that a hyperproperty is subset-closed, which are both crucial for the equivalence with the property-free criterion of **?**]. As in §2, we solve this problem by interpreting classes of properties as an *abstraction* of another class of properties induced by a closure operator. Differently from §2, the presence of adversarial contexts makes the criteria for subset-closed hyperproperties and trace properties distinct. **?**] show that the criterion for robust preservation of hypersafety is distinct from robust safety preservation and all criteria about classes of trace properties are distinct from their relational counterparts e.g., robust preservation of relational safety and robust preservation of safety properties are different. We therefore further generalize the argument from §3.2 to safety hyperproperties as well as to relational hyperproperties.

Specifically, we provide a trinity for the preservation of trace properties and subset-closed hyperproperties (§6.1), of safety properties and hypersafety hyperproperties (§6.2), of hyperproperties (§6.3), and for 2-relational (hyper)properties (§6.4). We conclude the section by studying the relative expressiveness of these criteria (§6.5).

Robustness and Compositional Compilation. Before diving into the criteria for robust compilation, it is worth noting the relationship between these and compositional compiler correctness. Compositional compiler correctness (*CCC*) is a statement of compiler correctness for programs that are linked against *some* contexts. Unlike robustness, which imposes no constraints on the contexts, *CCC* imposes conditions on the target contexts that compiled programs can be linked against: they need to be related (in ways that vary from work to work [**? ?**]) to the source contexts [**?**]. As **?**] also point out, the notions of *CCC* and of robust compilation are incomparable: neither can be proven stronger than the other. This is not surprising since robust compilation criteria are used to prove compiler security while *CCC* is used to prove correctness.[15]

The criteria we adopt could be generalised further by adding an extra parameter that qualifies the relation between source and target contexts. Such a general statement would let us express both *CCC* and robust compilation by picking the correct extra parameter. However, we refrain from presenting such general statements, as the implications in terms of preservation of classes of (hyper)properties has not been studied for them.

[15] We remark *CCC* has been used to conclude security of compilation in the previously discussed work of **?**] (and in its predecessor [**?**]). However, there is a key difference in the 'role' of contexts: in robust compilation criteria, contexts model attackers while in **?**] contexts are other bits of compiled code. This treatment lets **?**] reason compositionally about the concurrently-executing compiled code.

6.1 Trace-Relating Secure Compilation: Trace Properties and Subset-closed Hyperproperties

This section shows the simple generalization of RTC to the trace-relating setting (RTC$^{\sim}$) and its corresponding trinitarian view (Theorem 6.1). Then, it presents the trinitarian view for criteria that preserve subset-closed hyperproperties (Theorem 6.2).

Theorem 6.1 (Trinity for Robust Trace Properties 🐓). For any trace relation $\sim$ and induced property mappings $\tilde{\tau}$ and $\tilde{\sigma}$, we have: $\mathsf{RTP}^{\tilde{\tau}} \iff \mathsf{RTC}^{\sim} \iff \mathsf{RTP}^{\tilde{\sigma}}$, where

$$\begin{aligned}
\mathsf{RTC}^{\sim} &\equiv \forall \mathsf{P}\ \forall \mathsf{C_T}\ \forall \mathbf{t}.\ \mathsf{C_T}\left[\mathsf{P}\!\downarrow\right]\rightsquigarrow \mathbf{t} \Rightarrow \exists \mathsf{C_S}\ \exists \mathsf{s} \sim \mathbf{t}.\ \mathsf{C_S}\left[\mathsf{P}\right]\rightsquigarrow \mathsf{s}\\
\mathsf{RTP}^{\tilde{\tau}} &\equiv \forall \mathsf{P}\ \forall \pi_\mathsf{S} \in 2^{\mathsf{Trace_S}}.\ \mathsf{P} \models_\mathsf{R} \pi_\mathsf{S} \Rightarrow \mathsf{P}\!\downarrow\ \boldsymbol{\models}_\mathbf{R}\ \tilde{\tau}(\pi_\mathsf{S})\\
\mathsf{RTP}^{\tilde{\sigma}} &\equiv \forall \mathsf{P}\ \forall \boldsymbol{\pi}_\mathbf{T} \in 2^{\mathsf{Trace_T}}.\ \mathsf{P} \models_\mathsf{R} \tilde{\sigma}(\boldsymbol{\pi}_\mathbf{T}) \Rightarrow \mathsf{P}\!\downarrow\ \boldsymbol{\models}_\mathbf{R}\ \boldsymbol{\pi}_\mathbf{T}
\end{aligned}$$

The trinity for robust trace property preservation is the straightforward adaptation of the concepts of §2 to the definitions of ?]. Intuitively, these criteria simply deal with partial programs P instead of whole programs W. Necessarily, these criteria then consider arbitrary program contexts linked with P; the universal quantification over $\mathsf{C_S}$ and $\mathsf{C_T}$ are tacit in the expression $\models_R$.

We can also generalize §2 to *robust* subset-closed hyperproperties (Theorem 6.2). However, unlike the correct compilation case of §2, the equivalent property-free criterion (RSCHC$^{\sim}$) does not coincide with RSC$^{\sim}$, but states the existence of a single source context for all the target traces produced by a program in a given context.

Theorem 6.2 (Trinity for Robust Subset-closed Hyperproperties 🐓). Let $\mathsf{SCH_S}$ and $\mathsf{SCH_T}$ denote the sets of all subset-closed hyperproperties in the source and target languages, respectively. For any trace relation $\sim$ and its existential and universal images lifted to hyperproperties (that is, the lifting of the respective functions from Definition 2.5), $\tilde{\tau}$ and $\tilde{\sigma}$, and for $Cl_{\subseteq}(H) = \{\pi \mid \exists \pi' \in H.\ \pi \subseteq \pi'\}$, we have: $\mathsf{RSCHP}^{Cl_{\subseteq}\circ\tilde{\tau}} \iff \mathsf{RSCHC}^{\sim} \iff \mathsf{RSCHP}^{Cl_{\subseteq}\circ\tilde{\sigma}}$, where

$$\begin{aligned}
\mathsf{RSCHC}^{\sim} &\equiv \forall \mathsf{P}\ \forall \mathsf{C_T}\ \exists \mathsf{C_S}\ \forall \mathbf{t}\ \mathsf{C_T}\left[\mathsf{P}\!\downarrow\right]\rightsquigarrow \mathbf{t} \Rightarrow \exists \mathsf{s} \sim \mathbf{t}'.\ \mathsf{C_S}\left[\mathsf{P}\right]\rightsquigarrow \mathsf{s}\\
\mathsf{RSCHP}^{Cl_{\subseteq}\circ\tilde{\tau}} &\equiv \forall \mathsf{P}\ \forall \mathsf{H_S} \in \mathsf{SCH_S}.\ \mathsf{P} \models_\mathsf{R} \mathsf{H_S} \Rightarrow \mathsf{P}\!\downarrow\ \boldsymbol{\models}_\mathbf{R}\ Cl_{\subseteq}(\tilde{\tau}(\mathsf{H_S}))\\
\mathsf{RSCHP}^{Cl_{\subseteq}\circ\tilde{\sigma}} &\equiv \forall \mathsf{P}\ \forall \mathbf{H_T} \in \mathbf{SCH_T}.\ \mathsf{P} \models_\mathsf{R} Cl_{\subseteq}(\tilde{\sigma}(\mathbf{H_T})) \Rightarrow \mathsf{P}\!\downarrow\ \boldsymbol{\models}_\mathbf{R}\ \mathbf{H_T}
\end{aligned}$$

6.2 Trace-Relating Secure Compilation: Safety and Hypersafety

In this section we elaborate the robust preservation of safety (Theorem 6.3) and hypersafety properties (Theorem 6.4). Similar to §3.2, we consider the trace model adopted by ?] to ease the presentation. Our starting point is the two equivalent criteria for preservation of robust satisfaction of *all* and *only* the safety properties [?],

$$\begin{aligned}
\mathsf{RSP} &\equiv \forall \mathsf{P}.\ \forall \pi \in \mathit{Safety}.\ \mathsf{P}\models_\mathsf{R}\pi \Rightarrow \mathsf{P}\!\downarrow\models_\mathsf{R}\pi\\
\mathsf{RSC} &\equiv \forall \mathsf{P}.\ \forall \mathsf{C_T}.\forall m.\mathsf{C_T}\left[\mathsf{P}\!\downarrow\right]\rightsquigarrow^* m \Rightarrow \exists \mathsf{C_S}.\ \mathsf{C_S}\left[\mathsf{P}\right]\rightsquigarrow^* m
\end{aligned}$$

where $\mathsf{C_T}\left[\mathsf{P}\!\downarrow\right]\rightsquigarrow^* m$ is a shorthand for $\exists t \geq m.\mathsf{C_T}\left[\mathsf{P}\!\downarrow\right]\rightsquigarrow t$.

RSP differs from RTP as it only quantifies over safety properties, and RSC differs from RTC as it quantifies over finite prefixes m, rather than complete traces t. This comes from the fact that safety properties can be characterized in terms of sets of *bad* prefixes (as in Definition 3.4). Unfolding $\rightsquigarrow^*$ we can interpret RSC as follows. If $\mathsf{C_T}\left[\mathsf{P}\!\downarrow\right]$ produces a trace $t \geq m$ that violates a specific safety property, namely, the one defined by $M = \{m\}$, then there exists $\mathsf{C_S}$ in which P violates the *same* safety property, producing a trace $t' \geq m$ but possibly distinct from t.

Our generalization of RSC to the trace-relating setting states that whenever $\mathsf{C_T}\,[\mathsf{P}{\downarrow}]$ produces a trace t that violates a target safety property, there exists a source context $\mathsf{C_S}$ in which P violates the source *interpretation* of the property, i.e., its image through $\tilde{\sigma}$. The following theorem defines $\mathsf{RSC}^{\sim}$ and its two equivalent formulations.

Theorem 6.3 (Trinity for Robust Safety Properties). For any trace relation $\sim$ and for the corresponding property mappings $\tilde{\tau}$ and $\tilde{\sigma}$, we have: $\mathsf{RTP}^{\mathit{Safe}\circ\tilde{\tau}} \iff \mathsf{RSC}^{\sim} \iff \mathsf{RSP}^{\tilde{\sigma}}$, where

$$\begin{aligned}
\mathsf{RSC}^{\sim} &\equiv \forall \mathsf{P}\ \forall \mathsf{C_T}\ \forall \mathsf{t}\ \forall \mathsf{m} \leq \mathsf{t}. \mathsf{C_T}\,[\mathsf{P}{\downarrow}] {\rightsquigarrow} \mathsf{t} \Rightarrow \exists \mathsf{C_S}\ \exists \mathsf{t}' \geq \mathsf{m}\ \exists \mathsf{s} \sim \mathsf{t}'.\ \mathsf{C_S}\,[\mathsf{P}] {\rightsquigarrow} \mathsf{s} \\
\mathsf{RTP}^{\mathit{Safe}\circ\tilde{\tau}} &\equiv \forall \mathsf{P} \forall \pi_\mathsf{S} \in 2^{\mathsf{Trace_S}}. \mathsf{P} \models_\mathsf{R} \pi_\mathsf{S} \Rightarrow \mathsf{P}{\downarrow} \models_\mathsf{R} (\mathit{Safe} \circ \tilde{\tau})(\pi_\mathsf{S}) \\
\mathsf{RSP}^{\tilde{\sigma}} &\equiv \forall \mathsf{P} \forall \pi_\mathsf{T} \in \mathsf{Safety_T}. \mathsf{P} \models_\mathsf{R} \tilde{\sigma}(\pi_\mathsf{T}) \Rightarrow \mathsf{P}{\downarrow} \models_\mathsf{R} \pi_\mathsf{T}
\end{aligned}$$

where the closure operator *Safe* is the one introduced in §3.2.

Exactly like §3.2, Theorem 6.3 exploits the fact that

$$\mathit{Safe} \circ \tilde{\tau} : 2^{\mathsf{Trace_S}} \leftrightarrows \mathsf{Safety_T} : \tilde{\sigma}$$

is a Galois connection between source properties and target safety properties and the argument generalizes to arbitrary closure operators on target properties (). More interestingly, we can further generalize this idea to hypersafety. Hypersafety lifts the idea of safety with another level of sets (just like hyperproperties do w.r.t. trace properties) in order to talk about multiple runs of the same program. Just like for safety, hypersafety is concerned with a set of bad prefixes (called M) that no program upholding the hypersafety property should extend. Formally, a hyperproperty H is hypersafety if: $\forall \pi. \pi \notin H \Rightarrow (\exists M. M < \pi \wedge (\forall \pi' M < \pi' \Rightarrow \pi' \notin H))$. In Theorem 6.4, we indeed exploit the following Galois connection between source subset-closed hyperproperties and target

$$\mathit{HSafe} \circ \tilde{\tau} : \mathsf{SCH_S} \leftrightarrows \mathsf{HSafety_T} : \mathit{Cl}_{\subseteq} \circ \tilde{\sigma}$$

where $\mathsf{HSafety_T} = \left\{ \mathit{HSafe}(\mathsf{H_T}) \;\middle|\; \mathsf{H_T} \in 2^{2^{\mathsf{Trace_T}}} \right\}$ and *HSafe* is the closure operator that maps an arbitrary target hyperproperty $\mathsf{H_T}$ to the target hypersafety that best over-approximates $\mathsf{H_T}$.[16]

Theorem 6.4 (Trinity for Robust Hypersafety). For any trace relation $\sim$ and for the induced property mappings $\tilde{\tau}$ and $\tilde{\sigma}$, we have: $\mathsf{RSCHP}^{\mathit{HSafe}\circ\tilde{\tau}} \iff \mathsf{RHSC}^{\sim} \iff \mathsf{RHSP}^{\mathit{Cl}_{\subseteq}\circ\tilde{\sigma}}$, where

$$\begin{aligned}
\mathsf{RHSC}^{\sim} &\equiv \forall \mathsf{P}\ \forall \mathsf{C_T}\ \forall \mathsf{M} \in \mathcal{M}^{\mathit{fin}}.\ \mathsf{M} \leq \mathit{beh}(\mathsf{C_T}[\mathsf{P}{\downarrow}]) \Rightarrow \\
&\qquad \exists \mathsf{C_S}\ \forall \mathsf{m} \in \mathsf{M}\ \exists \mathsf{t} \geq \mathsf{m}.\ \exists \mathsf{s} \sim \mathsf{t}.\ \mathsf{C_S}\,[\mathsf{P}] {\rightsquigarrow} \mathsf{s} \\
\mathsf{RSCHP}^{\mathit{HSafe}\circ\tilde{\tau}} &\equiv \forall \mathsf{P}\ \forall \mathsf{H_S} \in \mathsf{SCH_S}.\ \mathsf{P} \models_\mathsf{R} \mathsf{H_S} \Rightarrow \mathsf{P}{\downarrow} \models_\mathsf{R} \mathit{HSafe}(\tilde{\tau}(\mathsf{H_S})) \\
\mathsf{RHSP}^{\mathit{Cl}_{\subseteq}\circ\tilde{\sigma}} &\equiv \forall \mathsf{P}\ \forall \mathsf{H_T} \in \mathsf{HSafety_T}. \mathsf{P} \models_\mathsf{R} \mathit{Cl}_{\subseteq}(\tilde{\sigma}(\mathsf{H_T})) \Rightarrow \mathsf{P}{\downarrow} \models_\mathsf{R} \mathsf{H_T}
\end{aligned}$$

and $\mathcal{M}^{\mathit{fin}}$ is the set of *finite* sets of prefixes.

We conclude this section with the following remark. The reader might wonder about extracting a “new” trace relation from the Galois connection $\mathit{Safe} \circ \tilde{\tau} : 2^{\mathsf{Trace_S}} \leftrightarrows \mathsf{Safety_T} : \tilde{\sigma}$ and get another formulation of $\mathsf{RSC}^{\sim}$. We note that this is not possible in general, as the class of safety properties, i.e., closed sets, is not necessarily a powerset and hence Lemma 2.7 cannot be applied.

[16] $\mathit{HSafe}(\mathsf{H_T}) = \cap \left\{ \mathsf{H'_T} \;\middle|\; \mathsf{H_T} \subseteq \mathsf{H'_T} \wedge \mathsf{H'_T} \in \mathsf{HSafety_T} \right\}$. See, e.g., ?] and ?].

6.3 Trace-Relating Secure Compilation: Arbitrary Hyperproperties

We already mentioned that some properties of interest for security e.g., possibilistic information-flow are not subset closed [**?**]. In this section we lift the results from §3.3 to the *secure* compilation setting. Once again, the trinity is *weak* as the equivalence to $\mathsf{RHP}^{\tilde{\sigma}}$ requires an extra assumption.

Theorem 6.5 (Weak Trinity for Robust Hyperproperties 🐓)**.** For a trace relation $\sim \subseteq \mathsf{Trace_S} \times \mathsf{Trace_T}$ and induced property mappings $\tilde{\sigma}$ and $\tilde{\tau}$, we have:
$\mathsf{RHC}^{\sim} \iff \mathsf{RHP}^{\tilde{\tau}}$;
if $\tilde{\tau} \leftrightarrows \tilde{\sigma}$ is a Galois insertion (i.e., $\tilde{\tau} \circ \tilde{\sigma} = id$), then $\mathsf{RHC}^{\sim} \Rightarrow \mathsf{RHP}^{\tilde{\sigma}}$,
if $\tilde{\sigma} \leftrightarrows \tilde{\tau}$ is a Galois reflection (i.e., $\tilde{\sigma} \circ \tilde{\tau} = id$), then $\mathsf{RHP}^{\tilde{\sigma}} \Rightarrow \mathsf{RHP}^{\tilde{\tau}}$,

$$\begin{aligned} \text{where } \mathsf{RHC}^{\sim} &\equiv \forall \mathsf{P}\ \forall \mathsf{C_T}\ \exists \mathsf{C_S}\ \forall \mathsf{t}.\ \mathsf{C_T}\left[\mathsf{P}\!\downarrow\right]\rightsquigarrow \mathsf{t} \iff (\exists \mathsf{s} \sim \mathsf{t}.\ \mathsf{C_S}\left[\mathsf{P}\right]\rightsquigarrow \mathsf{s}) \\ \mathsf{RHP}^{\tilde{\tau}} &\equiv \forall \mathsf{P}\ \forall \mathsf{H_S}.\ \mathsf{P} \models_{\mathsf{R}} \mathsf{H_S} \Rightarrow \mathsf{P}\!\downarrow\ \models_{\mathsf{R}} \tilde{\tau}(\mathsf{H_S}) \\ \mathsf{RHP}^{\tilde{\sigma}} &\equiv \forall \mathsf{P}\ \forall \mathsf{H_T}.\ \mathsf{P} \models_{\mathsf{R}} \tilde{\sigma}(\mathsf{H_T}) \Rightarrow \mathsf{P}\!\downarrow\ \models_{\mathsf{R}} \mathsf{H_T} \end{aligned}$$

It is therefore possible and correct to deduce a source obligation for a given target hyperproperty $\mathsf{H_T}$ ($\mathsf{RHC}^{\sim} \Rightarrow \mathsf{RHP}^{\tilde{\sigma}}$) when no information is lost in the composition $\tilde{\tau} \circ \tilde{\sigma}$. On the other hand, $\mathsf{RHP}^{\tilde{\tau}}$ is a consequence of $\mathsf{RHP}^{\tilde{\sigma}}$ when no information is lost in composing in the other direction, $\tilde{\sigma} \circ \tilde{\tau}$.

6.4 Trace-Relating Secure Compilation: 2-Relational Hyperproperties

Finally, we turn to *relational* properties and hyperproperties. Relational hyperproperties, as defined by **?**], are predicates on a sequence of behaviors; a sequence of programs has the relational hyperproperty if their behaviors collectively satisfy the predicate. Depending on the arity of the sequence, there exist different subclasses of relational hyperproperties, though for simplicity here we only study relational hyperproperties of arity 2. A key example of a relational hyperproperty is trace equivalence, which holds if two programs have identical behaviors.

All the trinities in this section follow the pattern of their non-relational counterparts. We first explain how one can get a Galois connection between source and target relational properties from a trace relation.

Given a trace relation $\sim \subseteq \mathsf{Trace_S} \times \mathsf{Trace_T}$, we can relate pairs of source traces with pairs of target traces point-wise,

$$(\mathsf{s_1}, \mathsf{s_2}) \sim (\mathsf{t_1}, \mathsf{t_2}) \iff \mathsf{s_1} \sim \mathsf{t_1} \wedge \mathsf{s_2} \sim \mathsf{t_2}$$

Formally this is $\sim^2 \subseteq {\mathsf{Trace_S}}^2 \times {\mathsf{Trace_T}}^2$, the product of the relation $\sim$ with itself. Therefore by Lemma 2.7 it corresponds to a Galois connection between source and target relational properties (🐓), that with a little abuse of notation[17] we still denote by

$$\tilde{\tau} : 2^{\mathsf{Trace_S} \times \mathsf{Trace_S}} \leftrightarrows 2^{\mathsf{Trace_T} \times \mathsf{Trace_T}} : \tilde{\sigma}$$

Explicitly, for $\mathsf{r_S} \in 2^{\mathsf{Trace_S} \times \mathsf{Trace_S}}$ and $\mathsf{r_T} \in 2^{\mathsf{Trace_T} \times \mathsf{Trace_T}}$,

$$\begin{aligned} \tilde{\tau}(\mathsf{r_S}) &= \{(\mathsf{t_1}, \mathsf{t_2}) \mid \exists (\mathsf{s_1}, \mathsf{s_2}).\ \mathsf{s_1} \sim \mathsf{t_1} \wedge \mathsf{s_2} \sim \mathsf{t_2} \wedge (\mathsf{s_1}, \mathsf{s_2}) \in \mathsf{r_S}\} \\ \tilde{\sigma}(\mathsf{r_T}) &= \{(\mathsf{s_1}, \mathsf{s_2}) \mid \forall (\mathsf{t_1}, \mathsf{t_2}).\ \mathsf{s_1} \sim \mathsf{t_1} \wedge \mathsf{s_2} \sim \mathsf{t_2} \Rightarrow (\mathsf{t_1}, \mathsf{t_2}) \in \mathsf{r_T}\} \end{aligned}$$

$\tilde{\tau}$ and $\tilde{\sigma}$ are then lifted to relational hyperproperties similarly to Definition 3.2. Explicitly, for $\mathsf{R_S} \in 2^{2^{\mathsf{Trace_S} \times \mathsf{Trace_S}}}$ and $\mathsf{R_T} \in 2^{2^{\mathsf{Trace_T} \times \mathsf{Trace_T}}}$,

$$\begin{aligned} \tilde{\tau}(\mathsf{R_S}) &= \{\tilde{\tau}(\mathsf{r_S}) \mid \mathsf{r_S} \in \mathsf{R_S}\} \\ \tilde{\sigma}(\mathsf{R_T}) &= \{\tilde{\sigma}(r_{\mathsf{T}}) \mid r_{\mathsf{T}} \in \mathsf{R_T}\} \end{aligned}$$

[17] Technically, we should write: $\tilde{\tau}^2 \leftrightarrows \tilde{\sigma}^2$

Given a relational property $r \in 2^{\mathit{Trace} \times \mathit{Trace}}$ and two programs P_1, P_2, we write $P_1, P_2 \models_R r$ for

$$\forall C.\ \forall t_1 t_2.\ C\,[P_1] \rightsquigarrow t_1 \wedge\ C\,[P_2] \rightsquigarrow t_2 \Rightarrow (t_1, t_2) \in r$$

Given a relational hyperproperty $R \in 2^{2^{\mathit{Trace} \times \mathit{Trace}}}$, by $P_1, P_2 \models_R R$ we mean

$$\forall C.(\mathit{beh}(C\,[P_1]), \mathit{beh}(C\,[P_2])) \in R$$

Theorem 6.6 (Trinity for Robust 2-Relational Trace Properties 🐓)**.** For any trace relation $\sim$ and for the corresponding property mappings $\tilde{\tau}$ and $\tilde{\sigma}$, we have: $\mathsf{R2rTP}^{\tilde{\tau}} \iff \mathsf{R2rTC}^{\sim} \iff \mathsf{R2rTP}^{\tilde{\sigma}}$, where

$$\begin{aligned}
\mathsf{R2rTC}^{\sim} \ \equiv\ & \forall \mathbf{C_T}\ \forall \mathsf{P_1}\ \forall \mathsf{P_2}\ \forall \mathbf{t_1}\ \forall \mathbf{t_2}.\ (\mathbf{C_T}\,[\mathsf{P_1}{\downarrow}] \rightsquigarrow \mathbf{t_1} \wedge \mathbf{C_T}\,[\mathsf{P_2}{\downarrow}] \rightsquigarrow \mathbf{t_2}) \Rightarrow \\
& \exists \mathsf{C_S}\ \exists \mathsf{s_1} \sim \mathbf{t_1}\ \exists \mathsf{s_2} \sim \mathbf{t_2}.\ \mathsf{C_S}\,[\mathsf{P_1}] \rightsquigarrow \mathsf{s_1} \wedge \mathsf{C_S}\,[\mathsf{P_2}] \rightsquigarrow \mathsf{s_2} \\
\mathsf{R2rTP}^{\tilde{\tau}} \ \equiv\ & \forall \mathsf{P_1 P_2}\ \forall \mathsf{r_S} \in 2^{\mathsf{Trace_S} \times \mathsf{Trace_S}}.\ \mathsf{P_1}, \mathsf{P_2} \models_{\mathsf{R}} \mathsf{r_S} \Rightarrow \quad \mathsf{P_1}{\downarrow}, \mathsf{P_2}{\downarrow} \boldsymbol{\models}_{\mathbf{R}} \tilde{\tau}(\mathsf{r_S}) \\
\mathsf{R2rTP}^{\tilde{\sigma}} \ \equiv\ & \forall \mathsf{P_1 P_2}.\ \forall \mathbf{r_T} \in 2^{\mathbf{Trace_T} \times \mathbf{Trace_T}}.\ \mathsf{P_1}, \mathsf{P_2} \models_{\mathsf{R}} \tilde{\sigma}(\mathbf{r_T}) \Rightarrow \quad \mathsf{P_1}{\downarrow}, \mathsf{P_2}{\downarrow} \boldsymbol{\models}_{\mathbf{R}} \mathbf{r_T}
\end{aligned}$$

Next, we propose the trinity for 2-relational subset-closed hyperproperties, i.e., elements of $2^{2^{\mathit{Trace} \times \mathit{Trace}}}$ that are closed under subsets. Exactly as in the case of subset-closed hyperproperties, the application of $\tilde{\tau}$ and $\tilde{\sigma}$ may lose the information of being subset-closed. In order to guarantee the equivalence of the three criteria, we compose the two mappings with a closure operator that we still denote by $\mathit{Cl}_{\subseteq}$.

Theorem 6.7 (Trinity for 2-Relational Robust Subset-Closed Hyperproperties 🐓)**.** For any trace relation $\sim$ and for the corresponding property mappings $\tilde{\tau}$ and $\tilde{\sigma}$, we have $\mathsf{R2rSCHP}^{\mathit{Cl}_{\subseteq} \circ \tilde{\tau}} \iff \mathsf{R2rSCHC}^{\sim} \iff \mathsf{R2rSCHP}^{\mathit{Cl}_{\subseteq} \circ \tilde{\sigma}}$, where

$$\begin{aligned}
\mathsf{R2rSCHC}^{\sim} \ \equiv\ & \forall \mathbf{C_T}\ \forall \mathsf{P_1}\ \forall \mathsf{P_2}\ \exists \mathsf{C_S}\ \forall \mathbf{t_1}\ \forall \mathbf{t_2}.\ (\mathbf{C_T}\,[\mathsf{P_1}{\downarrow}] \rightsquigarrow \mathbf{t_1} \wedge \mathbf{C_T}\,[\mathsf{P_2}{\downarrow}] \rightsquigarrow \mathbf{t_2}) \Rightarrow \\
& \exists \mathsf{s_1} \sim \mathbf{t_1}\ \exists \mathsf{s_2} \sim \mathbf{t_2}.\ \mathsf{C_S}\,[\mathsf{P_1}] \rightsquigarrow \mathsf{s_1} \wedge \mathsf{C_S}\,[\mathsf{P_2}] \rightsquigarrow \mathsf{s_2} \\
\mathsf{R2rSCHP}^{\mathit{Cl}_{\subseteq} \circ \tilde{\tau}} \ \equiv\ & \forall \mathsf{P_1}\ \forall \mathsf{P_2}\ \forall \mathsf{R_S} \in \mathsf{2RelSCH_S}.\ \mathsf{P_1}, \mathsf{P_2} \models_{\mathsf{R}} \mathsf{R_S} \Rightarrow \mathsf{P_1}{\downarrow}, \mathsf{P_2}{\downarrow} \boldsymbol{\models}_{\mathbf{R}} \tilde{\tau}(\mathsf{R_S}) \\
\mathsf{R2rSCHP}^{\mathit{Cl}_{\subseteq} \circ \tilde{\sigma}} \ \equiv\ & \forall \mathsf{P_1}\ \forall \mathsf{P_2}\ \forall \mathbf{R_T} \in \mathbf{2RelSCH_T}.\mathsf{P_1}, \mathsf{P_2} \models_{\mathsf{R}} \tilde{\sigma}(\mathbf{R_T}) \Rightarrow \mathsf{P_1}{\downarrow}, \mathsf{P_2}{\downarrow} \boldsymbol{\models}_{\mathbf{R}} \mathbf{R_T}
\end{aligned}$$

We move now to the class of relational safety properties, a notion that generalizes safety properties to relations on programs. Similarly to Theorem 6.3, $\mathsf{R2rSP}^{\tilde{\sigma}}$ quantifies over target relational safety properties, while $\mathsf{R2rTP}^{\mathit{2rSafe} \circ \tilde{\tau}}$ quantifies over all source relational property and compose $\tilde{\tau}$ with $\mathit{2rSafe}$ a closure operator that best approximates a relational property with a relational safety property.

Theorem 6.8 (Trinity for Robust 2-Relational Safety Properties 🐓)**.** For any trace relation $\sim$ and for the corresponding property mappings $\tilde{\tau}$ and $\tilde{\sigma}$, we have: $\mathsf{R2rTP}^{\mathit{2rSafe} \circ \tilde{\tau}} \iff \mathsf{R2rSC}^{\sim} \iff \mathsf{R2rSP}^{\tilde{\sigma}}$, where

$$\begin{aligned}
\mathsf{R2rSC}^{\sim} \ \equiv\ & \forall \mathbf{C_T}\ \forall \mathsf{P_1 P_2}\ \forall \mathbf{t_1 t_2}\ \forall \mathbf{m_1} \leq \mathbf{t_1}\ \forall \mathbf{m_2} \leq \mathbf{t_2}.\ \mathbf{C_T}\,[\mathsf{P_1}{\downarrow}] \rightsquigarrow \mathbf{t_1} \Rightarrow \mathbf{C_T}\,[\mathsf{P_2}{\downarrow}] \rightsquigarrow \mathbf{t_2} \Rightarrow \\
& \exists \mathsf{C_S}\ \exists \mathbf{t'_1} \geq \mathbf{m_1}\ \exists \mathsf{s_1} \sim \mathbf{t'_1}\ \exists \mathbf{t'_2} \geq \mathbf{m_2}\ \exists \mathsf{s_2} \sim \mathbf{t'_2}.\ \mathsf{C_S}\,[\mathsf{P_1}] \rightsquigarrow \mathsf{s_1} \wedge \mathsf{C_S}\,[\mathsf{P_2}] \rightsquigarrow \mathsf{s_2} \\
\mathsf{R2rTP}^{\mathit{2rSafe} \circ \tilde{\tau}} \ \equiv\ & \forall \mathsf{P_1 P_2}\ \forall \mathsf{r_S} \in 2^{\mathsf{Trace_S} \times \mathsf{Trace_S}}.\ \mathsf{P_1}, \mathsf{P_2} \models_{\mathsf{R}} \mathsf{r_S} \Rightarrow \mathsf{P_1}{\downarrow}, \mathsf{P_2}{\downarrow} \boldsymbol{\models}_{\mathbf{R}} (\mathit{2rSafe} \circ \tilde{\tau})(\mathsf{r_S}))) \\
\mathsf{R2rSP}^{\tilde{\sigma}} \ \equiv\ & \forall \mathsf{P_1 P_2}\ \forall \mathbf{r_T} \in \mathbf{2rel\text{-}Safety_T}.\ \mathsf{P_1}, \mathsf{P_2} \models_{\mathsf{R}} \tilde{\sigma}(\mathbf{r_T}) \Rightarrow \mathsf{P_1}{\downarrow}, \mathsf{P_2}{\downarrow} \boldsymbol{\models}_{\mathbf{R}} \mathbf{r_T}
\end{aligned}$$

Finally, we present the most general criterion: preservation of *arbitrary* 2-relational hyperproperties. As for the preservation of arbitrary hyperproperties, this (weak) trinity requires additional assumptions to hold, namely that the Galois connection is an insertion or a reflection.

Theorem 6.9 (Weak trinity for Robust 2-Relational Hyperproperties 🐓)**.** For a trace relation $\sim\ \subseteq \mathsf{Trace_S} \times \mathsf{Trace_T}$ and the corresponding property mappings $\tilde{\sigma}$ and $\tilde{\tau}$, we have:
$\mathsf{R2rHC}^{\sim} \iff \mathsf{R2rHP}^{\tilde{\tau}}$;
if $\tilde{\tau} \leftrightarrows \tilde{\sigma}$ is a Galois insertion (i.e., $\tilde{\tau} \circ \tilde{\sigma} = id$), then $\mathsf{R2rHC}^{\sim} \Rightarrow \mathsf{R2rHP}^{\tilde{\sigma}}$,
if $\tilde{\sigma} \leftrightarrows \tilde{\tau}$ is a Galois reflection (i.e., $\tilde{\sigma} \circ \tilde{\tau} = id$), then $\mathsf{R2rHP}^{\tilde{\sigma}} \Rightarrow \mathsf{R2rHP}^{\tilde{\tau}}$,

$$\begin{aligned}
\text{where } \mathsf{R2rHC}^{\sim} \equiv\ & \forall \mathsf{P_1 P_2}\ \forall \mathsf{C_T}\ \exists \mathsf{C_S}. \\
& (\forall \mathsf{t}.\ \mathsf{C_T}\,[\mathsf{P_1}\downarrow] \rightsquigarrow \mathsf{t} \iff (\exists \mathsf{s} \sim \mathsf{t}.\ \mathsf{C_S}\,[\mathsf{P_1}] \rightsquigarrow \mathsf{s})) \land \\
& (\forall \mathsf{t}.\ \mathsf{C_T}\,[\mathsf{P_2}\downarrow] \rightsquigarrow \mathsf{t} \iff (\exists \mathsf{s} \sim \mathsf{t}.\ \mathsf{C_S}\,[\mathsf{P_2}] \rightsquigarrow \mathsf{s})) \\
\mathsf{R2rHP}^{\tilde{\tau}} \equiv\ & \forall \mathsf{P_1}\ \forall \mathsf{P_2}\ \forall \mathsf{R_S}.\ \mathsf{P_1}, \mathsf{P_2} \models_{\mathsf{R}} \mathsf{R_S} \Rightarrow \mathsf{P_1}\downarrow, \mathsf{P_2}\downarrow \models_{\mathsf{R}} \tilde{\tau}(\mathsf{R_S}) \\
\mathsf{R2rHP}^{\tilde{\sigma}} \equiv\ & \forall \mathsf{P_1}\ \forall \mathsf{P_2}\ \forall \mathsf{R_T}. \mathsf{P_1}, \mathsf{P_2} \models_{\mathsf{R}} \tilde{\sigma}(\mathsf{R_T}) \Rightarrow \mathsf{P_1}\downarrow, \mathsf{P_2}\downarrow \models_{\mathsf{R}} \mathsf{R_T}
\end{aligned}$$

6.5 Relating the Secure Compilation Trinities

Figure 4 orders criteria referring to the same trace relation ~ according to their relative strength. If a trinity entails another (denoted by ⇒), then the former provides stronger security for a compilation chain than the latter.

The hypotheses of insertion and reflection mentioned in Theorem 6.9 and Theorem 6.5 are highlighted with the labels 'Ins' and 'Refl'. Recall that when composing $\tilde{\tau}$ with *Safe* we quantify over the whole class of source trace properties rather than only safety properties. This is represented by the blue background in $\mathsf{RTP}^{Safe \circ \tilde{\tau}}$. The trinity for the robust preservation of arbitrary trace properties is on the same blue background. Red and green backgrounds are reserved for subset-closed hyperproperties and arbitrary relational properties and serve the same purpose.

We now describe how to interpret the acronyms in Figure 4. All criteria start with R meaning they refer to robust preservation (secure compilation criteria). Criteria for relational hyperproperties—here only arity 2 is shown for simplicity—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 $\mathsf{RSCHC}^{\sim}$, *robust* (R) *two-relational* (2r) *safety-preserving* (S) *compilation* (C) is $\mathsf{R2rSC}^{\sim}$, etc.

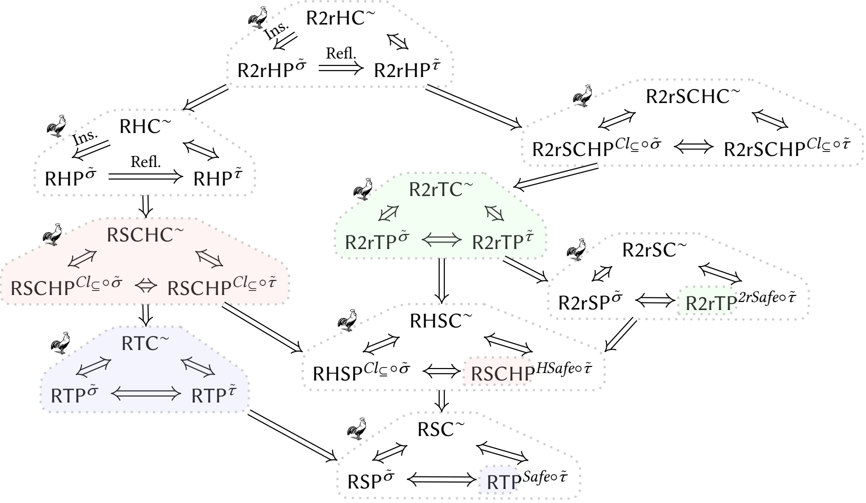
7 INSTANCES OF TRACE-RELATING SECURE COMPILATION

This section presents instances of compilers that adopt our framework for secure compilation purposes. We provide three illustrative cases, for compilers that respectively robustly-preserve trace properties (§7.1), safety properties (§7.2) and hypersafety properties (§7.3). The last two examples are not novel instances we devise but rather existing work whose results we recount as instantiations of our framework.

7.1 An Instance of Trace-Relating Robust Preservation of Trace Properties

This subsection illustrates trace-relating secure compilation when the target events are strictly more events than the source ones

The source and target languages used here extend the syntax of the source language of §4.3.1. Both languages have *outputs of naturals*, and the expressions that generate them: $\mathsf{out_S}\ \mathsf{n}$ and $\mathbf{out_S}\ \mathbf{e}$. Additionally, the target has a different output action and its related expression $\mathbf{out_T}\ \mathbf{n}$; this is the only difference between the languages. The extra events in the target model the ability of target language to perform potentially-dangerous operations (e.g., writing to the hard drive), which cannot


$\text{R2rHC}^{\sim}$
Ins.
$\text{R2rHP}^{\tilde{\sigma}}$
Refl.
$\text{R2rHP}^{\tilde{\tau}}$
$\text{R2rSCHC}^{\sim}$
$\text{R2rSCHP}^{Cl_{\subseteq}\circ\tilde{\sigma}}$
$\text{R2rSCHP}^{Cl_{\subseteq}\circ\tilde{\tau}}$
$\text{RHC}^{\sim}$
Ins.
$\text{RHP}^{\tilde{\sigma}}$
Refl.
$\text{RHP}^{\tilde{\tau}}$
$\text{R2rTC}^{\sim}$
$\text{R2rTP}^{\tilde{\sigma}}$
$\text{R2rTP}^{\tilde{\tau}}$
$\text{RSCHC}^{\sim}$
$\text{RSCHP}^{Cl_{\subseteq}\circ\tilde{\sigma}}$
$\text{RSCHP}^{Cl_{\subseteq}\circ\tilde{\tau}}$
$\text{R2rSC}^{\sim}$
$\text{R2rSP}^{\tilde{\sigma}}$
$\text{R2rTP}^{2rSafe\circ\tilde{\tau}}$
$\text{RHSC}^{\sim}$
$\text{RHSP}^{Cl_{\subseteq}\circ\tilde{\sigma}}$
$\text{RSCHP}^{HSafe\circ\tilde{\tau}}$
$\text{RTC}^{\sim}$
$\text{RTP}^{\tilde{\sigma}}$
$\text{RTP}^{\tilde{\tau}}$
$\text{RSC}^{\sim}$
$\text{RSP}^{\tilde{\sigma}}$
$\text{RTP}^{Safe\circ\tilde{\tau}}$

R robust	2r 2-relational	
H hyperproperties	SCH subset-closed hyperproperties	HS hypersafety
T trace properties	S safety properties	
C property-full criterion	P property-free criterion based on σ and τ	

Fig. 4. 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 trinities proven in Coq.

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 P, contexts C and linking of those two to produce whole programs $C[P]$. In this setting, a whole program W is the combination of a *main expression* to be evaluated and a set of *function definitions fs* (with distinct names) that can refer to their argument (*arg*) symbolically and can be called by the main expression and by other functions ($f(e)$. 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.

$$
\begin{array}{ll}
\mathsf{e} ::= \cdots \mid \mathsf{f(e)} \mid \mathsf{out_S\ n} \mid \mathsf{arg} & \mathbf{e} ::= \cdots \mid \mathbf{f(e)} \mid \mathbf{out_S\ n} \mid \mathbf{arg} \mid \mathbf{out_T\ n} \\
\mathsf{i} ::= \cdots \mid \mathsf{out_S\ n} & \mathbf{i} ::= \cdots \mid \mathbf{out_S\ n} \mid \mathbf{out_T\ n} \\
fs ::= \langle f_1, e_1 \rangle, \ldots, \langle f_n, e_n \rangle & P ::= \langle fs, e \rangle \qquad C ::= \ fs \qquad W ::= \ C[P]
\end{array}
$$

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 §4.3: it consists of a list of *regular events* (including the new outputs) terminated by a *result event*[18]. A partial program and a context can be linked into a whole program when their functions satisfy the requirements mentioned above.

[18] Notice that the languages are strictly terminating.

We define the homomorphic compiler ($\cdot\downarrow$) that translates each source construct into its target correspondent. Thus, the compiler never introduces the additional target instruction $\mathbf{out_T\ n}$. Since it is straightforward, the formalisation of the compiler is elided.

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 a given target trace t the source trace that erases from t those events that exist only at the target level. This is reasonable because only target contexts C (not compiled programs $P\downarrow$) can perform the extra target actions as the compiler does not introduce them. Let $t|_{\Sigma_S}$ indicate trace t filtered to retain only those elements included in alphabet Σ_S. We define the trace relation as:

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

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 this relation are:

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

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 the languages are very similar, it is simple to prove that our compiler is secure according to the trace relation $\sim$ defined above.

Theorem 7.1 ($\cdot\downarrow$ is Secure 🐓). $\cdot\downarrow$ *is* RTC$^{\sim}$.

7.2 An Instance of 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 [? ? ? ?]. 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 [?]. 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!).

?] consider precisely such a setting. Their languages are simple like those presented here but impure; their source has an ML-like heap and the target has a memory that is indexed by natural numbers and capabilities to protect addresses. Moreover, they define a compiler that preserves safety properties of source programs (i.e., it is RSC$^{\sim}$ in the sense of Theorem 6.3) by relying on the target capabilities. The interesting point, however, is that they also consider source and target traces to be distinct since the two languages have different values. Concretely, the source has bools and nats and the target only has nats, plus in the source, heap addresses are abstract locations ℓ while in the target they are nats. Thus, to prove RSC$^{\sim}$, they rely on a cross-language relation on values, which is lifted to trace actions, and then lifted point-wise to traces (analogously to what we have done in Sections 4.3, 4.4 and 7.1). In order to relate addresses, their cross-language relation is equipped with a partial bijection between source and target addresses, this bijection grows monotonically with every reduction step.

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.[19] 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. Thus, their compiler is proven to generate code that respects τ, so they really achieve a variation of $\mathsf{RTP}^{\mathit{Safe}\circ\tilde{\tau}}$ from Theorem 6.3. Their proofs are based on standard techniques either for secure compilation (i.e., trace-based backtranslation [?]) and for correct compilation (i.e., forward/backward simulation [?]).

7.3 An Instance of Trace-Relating Robust Preservation of Hypersafety Properties

?] 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 s and t respectively. The set of target output actions $\alpha!$ includes an action $\surd$ that has no source counterpart (i.e., $\nexists\alpha? \sim \surd$), and whose output does not depend on internal state (thus it cannot leak secrets).[20] By emitting $\surd$ whenever undesired inputs are fed to a compiled program (e.g., passing a nat when a bool is expected), hypersafety is preserved (as $\surd$ does not leak secrets) [?].

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 $\surd$) as follows:

$$\mathsf{TPC}\,(\pi_S) = \left\{ \mathsf{t} \;\middle|\; \mathsf{t} \in \bigcup_{n\in\mathbb{N}} \mathtt{int}_{n}\,(\pi_S) \right\} \quad \text{where } \mathtt{int}_{0}\,(\pi_S) = \{ \mathsf{t} \mid \exists \mathsf{s} \in \pi_S \land \mathsf{s} \sim \mathsf{t} \}$$

$$\mathtt{int}_{n+1}\,(\pi_S) = \{ \mathsf{t} \mid \mathsf{t} \equiv \mathsf{t}_1 \alpha? \surd \mathsf{t}_2 \land \mathsf{t}_1\mathsf{t}_2 \in \mathtt{int}_{n}\,(\pi_S) \land \mathtt{undesired}\,(\alpha?) \}$$

where $\mathtt{undesired}\,(\alpha?)$ indicates that $\alpha?$ is an undesired input (intuitively, this is an information that can be derived from the set of source traces [?]).

Informally, given a set of source traces π_S, TPC generates all target traces that are related (pointwise) to a source trace (case $\mathtt{int}_0$). Then (case $\mathtt{int}_{n+1}$), it adds all traces (t) with interleavings of undesired input $\alpha?$ (third conjunct) followed by $\surd$ (first conjunct) as long as the interleavings split a trace $\mathsf{t}_1\mathsf{t}_2$ that has already been mapped (second conjunct).

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.

8 RELATED WORK

We already discussed how our results relate to some existing work in correct compilation [? ?] and secure compilation [? ? ?]. We also already mentioned that most of our definitions and results make no assumptions about the structure of traces. One result that partially relies on the structure of traces is Theorem 6.3, that refers to *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 [?]. Without a notion of finite prefix only $\mathsf{RSC}^{\sim}$ may look different but both $\mathsf{RTP}^{\mathit{Safe}\circ\tilde{\tau}}$ and $\mathsf{RSP}^{\tilde{\sigma}}$ are

[19] They call those safety properties monitors since they focus on safety [?] and indicate s with M and t with M.

[20] Technically, they assume a set of $\surd$ actions, but for this analogy a single action suffices.

trace agnostic as in general safety properties can be defined as the closed sets of any topology on traces [?].

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* [?]. In the latter case we believe that the compilation from IMP to ASM proposed by **?**] 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, **?**] 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 §4.1 applies there too. This and many other verified compilers [? ? ? ?] beyond CakeML [?] deal with resource exhaustion and it would be interesting to also apply the ideas of §4.2 to them. [CH: There is more related work on resource exhaustion that we should be relating at some point: (1) CompCertTSO [?] and Gil's PLDI'15 [?] paper on Integer-Pointer casts. From [?]: "Therefore, CompCertTSO [?] models out of memory as the empty set of behaviors, that is no behavior. However, what happens if there were I/O events prior to the out-of-memory error? Discarding I/O events before running out of memory is absurd: the target program should always perform a prefix of the events the source program could have performed. To handle this, CompCert-TSO also observes partial behaviors, possibly before discarding behavior due to running out of memory: 4. A partial execution of the program that has produced a finite sequence of I/O events, e1, . . . , en, partial. As before, refinement is defined as inclusion of the set of (possibly partial) behaviors of the target program into that of the source program. In this paper, we follow CompCertTSO's approach of handling out of memory. However, unlike CompCertTSO, where only the target language can run out of memory, our source language can also run out of memory due to pointer-to-integer casts, as we explain below." (2) CompCertMC [?] also deals with memory exhaustion (at least for the stack), but I don't think we ever understood how their top level theorem looks like? From what I still remember they push low-level memory information to the higher levels via an oracle "stackspace" that is produced by the compiler together with the binary, an idea they credit to [?] (3) Mullen [?] also deal with a flat finite address space, so they must also have a solution for running out of memory. Andrew's note says they also use oracles.] [JT: Should we talk about this here, or in the related work section? Not sure we can keep this section short if we want to discuss these papers]

?] 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$^{\sim}$.[CH: Theirs is a *compositional* compiler result though.] [CH: Yale (NI preserving) [?]]

Switching to more informative traces capturing the interaction between the program and the context is often used as a proof technique for secure compilation [? ? ?]. Most of these results consider a cross-language relation, so they probably could be proved to attain one of the criteria from Figure 4.

[CH: also compositional compiler correctness works, not just Figure 2]

Generalizations of Compiler Correctness. The compiler correctness definition of **?**] 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 5 (left). And even some of the more recent compiler correctness definitions preserve this kind of flexibility [?]. While CC$^{\sim}$ can be seen as an instance of a definition by **?**], we are not aware of any prior work that investigated the preservation of properties when the "decode

$$\begin{tikzcd}
\text{source language} \arrow[r, "\text{source semantics}"] \arrow[d, "\text{compile}"'] & \text{source meanings} \\
\text{target language} \arrow[r, "\text{target semantics}"] & \text{target meanings} \arrow[u, "\text{decode}"']
\end{tikzcd}$$

$$\begin{tikzcd}
\mathsf{W} \arrow[r, two heads, "\mathsf{S}"] \arrow[d] & \mathsf{Z}^{\#} \\
\mathsf{W}\downarrow \arrow[r, two heads, "\mathsf{T}"] & \mathsf{Z} \arrow[u]
\end{tikzcd}$$

Fig. 5. ? ’s [?] (left) and ? ’s [?] and ? ’s [?] (right) compiler correctness diagrams

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. ?] and ?] expressed a strong variant of compiler correctness using the diagram of Figure 5 (right). 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 ↓ ($W \leq W' \iff W \twoheadrightarrow W'$ for both source and target). The “parallel” intuition can be formalized as an instance of $\mathsf{CC}^{\sim}$. Take source and target traces to be finite or infinite sequences of program states (maximal trace semantics [?]), and relate them exactly like ?] and ?].

Translation Validation. Translation validation is an important alternative to proving that all runs of a compiler are correct as it can be more easily applied to realistic compilers. An interesting work about translation validation of security properties has been recently proposed by ?]. They can handle many security properties expressible in terms of automata as long as source and target attackers and the observable traces are the same.
Instantiating the definition of any of the presented criteria with a particular program, one has translation validation criteria with the map $\tilde{\tau}$ describing the target property that is (robustly) satisfied once the translation is validated. For example one can consider

$$(\mathit{tsv}^{\sim})\ \mathsf{CC}^{\sim}(\mathsf{W}\downarrow) = \forall \mathsf{t}.\ \mathsf{W}\downarrow \rightsquigarrow \mathsf{t} \Rightarrow \exists \mathsf{s} \sim \mathsf{t}.\ \mathsf{W} \rightsquigarrow \mathsf{s}$$
$$(\mathit{rtsv}^{\sim})\ \mathsf{RTC}^{\sim}(\mathsf{C_T}\,[\mathsf{P}\downarrow]) = \forall \mathsf{t}.\ \mathsf{C_T}\,[\mathsf{P}\downarrow] \rightsquigarrow \mathsf{t} \Rightarrow \exists \mathsf{C_S}.\ \exists \mathsf{s} \sim \mathsf{t}.\ \mathsf{C_S}\,[\mathsf{P}] \rightsquigarrow \mathsf{s}$$

While the proof technique proposed by ?] might be generalized for $\mathsf{CC}^{\sim}(\mathsf{W}\downarrow)$ – as long as $\mathit{beh}(\mathsf{W}\downarrow)$ and $\mathit{beh}(\mathsf{W})$ can be expressed as one of the automata they can handle – they don’t work for $\mathsf{RTC}^{\sim}(\mathsf{C_T}\,[\mathsf{P}\downarrow])$ because of the existential in the conclusion.

?] are instead considering translation validation criteria in the spirit of $(\mathit{rtsv}^{\sim})$, their preliminary work only allows equality as trace relation, but should be subject to a generalization to the trace relating setting similar to the one we presented in this work.

Proof Techniques. We believe existing proof techniques (beyond the simulations discussed in Section 4.3.2) that have been devised to prove compiler correctness can also be employed to prove that a compiler attains any of the presented criteria. For example, cross-language binary logical relations can be used to relate two terms of two different languages when they ‘behave the same’ [? ? ?]. Additionally, they can also be used when multiple programs ‘behave the same’ [?] in a multilanguage semantics setting [?]. Secure compilation results (which rely on the criteria of Section 6) can be proven using variations of the *backtranslation* proof technique [? ? ?]. Presenting this proof techniques is beyond the scope of this paper, so we refer the interested reader to the work of ?].

9 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 $\mathsf{CC}^{\sim}$ 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?

Composition for Multipass Compilers. For now, we can already informally argue about the correctness of a multipass compiler, where each step is proved correct for a possibly different trace relation. Concretely, assume $\downarrow^{\mathsf{S}}_{\mathtt{I}}$ is a compilation chain from a source language S to an intermediate language $\mathtt{I}$ and $\downarrow^{\mathtt{I}}_{\mathbf{T}}$ from the intermediate language $\mathtt{I}$ to a target language $\mathbf{T}$.[21] Assume given two relations between traces of these languages: $\sim_{\mathsf{S},\mathtt{I}} \subseteq \mathsf{Trace}_{\mathsf{S}} \times \mathtt{Trace}_{\mathtt{I}}$ and $\sim_{\mathtt{I},\mathbf{T}} \subseteq \mathtt{Trace}_{\mathtt{I}} \times \mathbf{Trace}_{\mathbf{T}}$, such that each compiler is proven to be *CC* w.r.t. the expected trace relation: $\downarrow^{\mathsf{S}}_{\mathtt{I}} \in \mathsf{CC}^{\sim_{\mathsf{S},\mathtt{I}}}$ and $\downarrow^{\mathtt{I}}_{\mathbf{T}} \in \mathsf{CC}^{\sim_{\mathtt{I},\mathbf{T}}}$.

Let us consider the source-to-target compiler $\downarrow^{\mathsf{S}}_{\mathbf{T}}$ that is derived of the composition of the two aforementioned compilers, so $\downarrow^{\mathsf{S}}_{\mathbf{T}} = \downarrow^{\mathtt{I}}_{\mathbf{T}} \circ \downarrow^{\mathsf{S}}_{\mathtt{I}}$. In this case, we obtain the expected result: the correctness of the whole compiler $\downarrow^{\mathsf{S}}_{\mathbf{T}}$ is derived from the individual compiler correctness proofs for each step.

$$\mathsf{CC}^{(\sim_{\mathtt{I},\mathbf{T}} \circ \sim_{\mathsf{S},\mathtt{I}})} \equiv \forall \mathsf{W} \forall \mathbf{t}.\ \mathsf{W}\downarrow^{\mathsf{S}}_{\mathbf{T}} \rightsquigarrow \mathbf{t} \Rightarrow \exists \mathsf{s} \sim_{\mathtt{I},\mathbf{T}} \circ \sim_{\mathsf{S},\mathtt{I}} \mathbf{t}.\ \mathsf{W} \rightsquigarrow \mathsf{s}$$

where $\mathsf{s} \sim_{\mathsf{i},\mathbf{t}} \circ \sim_{\mathsf{s},\mathsf{i}} \mathbf{t} \iff \exists \mathtt{i} \in \mathtt{Trace}_{\mathtt{I}}.\ \mathsf{s} \sim_{\mathsf{S},\mathtt{I}} \mathtt{i} \land \mathtt{i} \sim_{\mathtt{I},\mathbf{T}} \mathbf{t}$.

Generalising this kind of composition to compilers that attain different criteria is unclear. For example, if $\downarrow^{\mathsf{S}}_{\mathtt{I}}$ preserves arbitrary hyperproperties, but $\downarrow^{\mathtt{I}}_{\mathbf{T}}$ preserves 2-relational safety properties, what can we conclude for $\downarrow^{\mathsf{S}}_{\mathbf{T}}$? We leave investigating these interesting matters for future work.

ACKNOWLEDGMENTS

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 CISPA-Stanford Center for Cybersecurity (FKZ: 13N1S0762), by DARPA grant SSITH/HOPE (FA8650-15-C-7558), by the Office of Naval Research for support through grant N00014-18-1-2620, Accountable Protocol Customization, and by UAIC internal grant 07/2018.

A PROOFS

Proof of Theorem 2.6 (🐓). See Theorems rel_TC_τTP and rel_TC_σTP in `TraceCriterion.v`, where the $\mathsf{TP}^{\tilde{\tau}} \iff \mathsf{TP}^{\tilde{\sigma}}$ part follows directly from Theorem 2.4. □

Proof of Lemma 2.7 (Trace relations ≅ Galois connections on trace properties).

?] 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 $\sim \subseteq X \times Y \mapsto \tilde{\tau} : 2^X \rightarrow 2^Y$.

[21] For the intermediate language we use a `verbatim`, `emerald` font.

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 $\tilde{\tau}$ (🐓), hence $\sim \mapsto \tilde{\tau} \leftrightarrows \tilde{\sigma}$ is itself bijective. □

Proof of Theorem 2.8 (Correspondence of Criteria). For a trace relation $\sim$ and the Galois connection $\tilde{\tau} \leftrightarrows \tilde{\sigma}$, the result follows from Theorem 2.6. For a Galois connection $\tau \leftrightarrows \sigma$ 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. □

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 $\alpha \leftrightarrows \gamma$ 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 4.3 (🐓). See Theorem correctness in `TypeRelationExampleInput.v`. □

Proof for Lemma 4.4 (gensend (·, ·) works). We proceed by induction on τ and then by induction on τ':

$\tau = \mathsf{N}$ **and** $\tau' = \mathsf{N}$ By canonicity we have that $\mathsf{r} = \langle \mathsf{n}, \mathsf{n}' \rangle$.
gensend (·, ·) translates that into **send n; send n′**.
By Rule Sem-seq, that produces $\mathbf{t} = \mathbf{n}; \mathbf{n}'$.
We need to prove that $\langle \mathsf{n}, \mathsf{n}' \rangle \sim \mathbf{n}; \mathbf{n}'$, which holds by Rule Trace-Rel-N-N.

$\tau = \mathsf{N}$ **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** $\tau' = \mathsf{N}$ Analogous to the other cases, by IH and Rule Trace-Rel-M-N.

$\tau = \tau_1 \times \tau_1'$ **and** $\tau' = \tau_2 \times \tau_2'$ So by canonicity $\mathsf{r} = \left\langle \left\langle \mathsf{r}_1, \mathsf{r}_1' \right\rangle, \left\langle \mathsf{r}_2, \mathsf{r}_2' \right\rangle \right\rangle$.
By definition of gensend (·, ·):

$$\begin{aligned} &\mathsf{gensend}\,(\mathbf{x}, \tau \times \tau') \\ &= \mathsf{gensend}\,(\mathbf{x}, \tau).\mathbf{1}; \mathsf{gensend}\,(\mathbf{x}, \tau').\mathbf{2} \end{aligned}$$

By the target reductions we know $(\mathsf{gensend}\,(\mathbf{x}, \tau).\mathbf{1}; \mathsf{gensend}\,(\mathbf{x}, \tau').\mathbf{2})[\mathsf{r}/\mathbf{x}] \rightsquigarrow \mathbf{is}_1; \mathbf{is}_2$, so by IH we have $\left\langle \mathsf{r}_1, \mathsf{r}_1' \right\rangle \sim \mathbf{is}_1$ and $\left\langle \mathsf{r}_2, \mathsf{r}_2' \right\rangle \sim \mathbf{is}_2$.
We need to prove that $\left\langle \left\langle \mathsf{r}_1, \mathsf{r}_1' \right\rangle, \left\langle \mathsf{r}_2, \mathsf{r}_2' \right\rangle \right\rangle \sim \mathbf{is}_1; \mathbf{is}_2$, which holds by Rule Trace-Rel-M-M, for $\mathsf{i} = \left\langle \mathsf{r}_1, \mathsf{r}_1' \right\rangle$ and $\mathsf{i}' = \left\langle \mathsf{r}_2, \mathsf{r}_2' \right\rangle$. □

Proof of Theorem 4.5. Trivial induction on the typing derivation of e, the only interesting case is the compilation of send e in the inductive cases.

Inductive. $\mathsf{e} = \mathsf{send\ e}$ By IH we have that if $(\vdash \mathsf{e} : \tau \times \tau')\!\downarrow\ \rightsquigarrow \mathbf{t}$ then $\exists \mathsf{s} \sim \mathbf{t}$ and $\mathsf{e} \rightsquigarrow \mathsf{t}$.
By definition of $(\cdot)\!\downarrow$ and of $\rightsquigarrow$ we need to prove that if

$$\mathbf{let\ x}{=}\,(\vdash \mathsf{e} : \tau \times \tau')\!\downarrow\ \mathbf{in}\ \mathsf{gensend}\,(\mathbf{x}, \tau \times \tau') \rightsquigarrow \mathbf{t}$$

Then $\mathsf{send\ e} \rightsquigarrow \mathsf{s}$ and $\mathsf{s} \sim \mathbf{t}$.
The reductions proceed as follows in the target:

$$\frac{(\vdash \mathsf{e} : \tau \times \tau')\!\downarrow\ \rightsquigarrow \langle \mathbf{is}, (\vdash \mathsf{r} : \tau \times \tau')\!\downarrow\rangle \qquad \mathsf{gensend}\,(\mathbf{x}, \tau \times \tau')[(\vdash \mathsf{r} : \tau \times \tau')\!\downarrow / \mathbf{x}] \rightsquigarrow \langle \mathbf{is}', \mathbf{r}'\rangle}{\mathbf{let}\ \mathbf{x}{=}\,(\vdash \mathsf{e} : \tau \times \tau')\!\downarrow\ \mathbf{in}\ \mathsf{gensend}\,(\mathbf{x}, \tau \times \tau') \rightsquigarrow \langle \mathbf{is} \cdot \mathbf{is}', \mathbf{r}'\rangle}$$

In the source we have $\dfrac{\mathsf{e} \rightsquigarrow \langle \mathsf{is}, \mathsf{r}\rangle}{\mathsf{send}\ \mathsf{e} \rightsquigarrow \langle \mathsf{is} \cdot \mathsf{r}, \mathsf{r}\rangle}$

By IH we have that $\mathsf{is} \sim \mathbf{is}$.

By Rule Trace-Rel-Single, to prove that $\mathsf{is}; \mathsf{r} \sim \mathbf{is}; \mathbf{is}'$ we need to prove that $\mathsf{is} \sim \mathbf{is}'$.

By Lemma 4.4 ($\mathsf{gensend}\,(\cdot, \cdot)$ works) we have that $\mathsf{r} \sim \mathbf{is}'$, so this case holds. □

Proof of Theorem 5.1. First of all we show that $\phi^{\#}$ is an *uco*, the proof for $\rho^{\#}$ is the same.

Monotonicity. $\phi^{\#}$ is composition of monotonic functions, hence it is itself monotonic.

Idempotence. We have to show that for π_{T}, $\phi^{\#}(\phi^{\#}(\pi_{\mathrm{T}})) = \phi^{\#}(\pi_{\mathrm{T}})$, that unfolding the definition means

$$g^{\circ} \circ \phi \circ f^{\circ} \circ g^{\circ} \circ \phi \circ f^{\circ}(\pi_{\mathrm{T}}) = g^{\circ} \circ \phi \circ f^{\circ}(\pi_{\mathrm{T}})$$

For the inclusion "⊆",

$$g^{\circ} \circ \phi \circ f^{\circ} \circ g^{\circ} \circ \phi \circ f^{\circ}(\pi_{\mathrm{T}}) \subseteq g^{\circ} \circ \phi \circ \phi \circ f^{\circ}(\pi_{\mathrm{T}}) = g^{\circ} \circ \phi \circ f^{\circ}(\pi_{\mathrm{T}})$$

the inclusion holds because $f^{\circ} \circ g^{\circ}(x) \subseteq x$ and the equality comes from idempotency of ϕ.

For the inclusion "⊇",

$$g^{\circ} \circ \phi \circ f^{\circ} \circ g^{\circ} \circ \phi \circ f^{\circ}(\pi_{\mathrm{T}}) \supseteq g^{\circ} \circ \phi \circ f^{\circ} \circ g^{\circ} \circ f^{\circ}(\pi_{\mathrm{T}}) = g^{\circ} \circ \phi \circ f^{\circ}(\pi_{\mathrm{T}})$$

the inclusion comes from $\phi(f^{\circ}(\pi_{\mathrm{T}})) \supseteq f^{\circ}(\pi_{\mathrm{T}})$ by extensiveness of ϕ, and the equality from $f^{\circ} \circ g^{\circ} \circ f^{\circ} = f^{\circ}$.

Extensiveness. We have to show that $\pi^{\#}(\pi_{\mathrm{T}}) \supseteq \pi_{\mathrm{T}}$.

$$\pi^{\#}(\pi_{\mathrm{T}}) = g^{\circ} \circ \phi \circ f^{\circ}(\pi_{\mathrm{T}}) \supseteq g^{\circ} \circ f^{\circ}(\pi_{\mathrm{T}}) \supseteq \pi_{\mathrm{T}}$$

The first inclusion is due to extensiveness of ϕ, the second by g° being the upper adjoint of f°.

For the statement of the theorem to hold, assume $\mathsf{W}{\models}ANI^{\rho}_{\phi}$ and $\mathsf{W}\!\downarrow\rightsquigarrow \mathbf{t}_1, \mathbf{t}_2$ with $\phi^{\#}(\mathbf{t}^{\circ}_1) = \phi^{\#}(\mathbf{t}^{\circ}_2)$, we have to show that $\rho^{\#}(\mathbf{t}^{\bullet}_1) = \rho^{\#}(\mathbf{t}^{\bullet}_1)$.
By CC$^{\sim}$ there exists $\mathsf{s}_1 \sim \mathbf{t}_1$ and $\mathsf{s}_2 \sim \mathbf{t}_2$ such that $\mathsf{W} \rightsquigarrow \mathsf{s}_1, \mathsf{s}_2$. As a preliminary, apply Lemma A.1 to the relations $\overset{\circ}{\sim} \circ$ *swap* and deduce g° is injective. Notice also that by functionality and totality, of $\overset{\circ}{\sim}$ and of $\overset{\bullet}{\sim}$, $f^{\circ}(\mathbf{t}^{\circ}_1) = \{\mathsf{s}^{\circ}_1\}$ and $f^{\bullet}(\mathbf{t}^{\bullet}_1) = \{\mathsf{s}^{\bullet}_1\}$ and a similar fact holds for s_2 and $\mathbf{t}_2$.

$$\begin{array}{ll}
\phi^{\#}(\mathbf{t}^{\circ}_1) = \phi^{\#}(\mathbf{t}^{\circ}_2) \Rightarrow & [\text{ definition of } \phi^{\#}] \\
g^{\circ} \circ \phi \circ f^{\circ}(\mathbf{t}^{\circ}_1) = g^{\circ} \circ \phi \circ f^{\circ}(\mathbf{t}^{\circ}_2) \Rightarrow & [g^{\circ} \text{ injective}] \\
\phi \circ f^{\circ}(\mathbf{t}^{\circ}_1) = \phi \circ f^{\circ}(\mathbf{t}^{\circ}_2) \Rightarrow & [f^{\circ}(\mathbf{t}^{\circ}_{\mathrm{i}}) = \mathsf{s}^{\circ}_{\mathrm{i}}\ i = 1, 2] \\
\phi(\mathsf{s}^{\circ}_1) = \phi(\mathsf{s}^{\circ}_2) \Rightarrow & [\mathsf{W}{\models}ANI^{\rho}_{\phi}] \\
\rho(\mathsf{s}^{\bullet}_1) = \rho(\mathsf{s}^{\bullet}_2) \Rightarrow & [\mathsf{s}^{\bullet}_{\mathrm{i}} = f^{\bullet}(\mathbf{t}^{\bullet}_{\mathrm{i}})\ i = 1, 2] \\
\rho \circ f^{\bullet}(\mathbf{t}^{\bullet}_1) = \rho \circ f^{\bullet}(\mathbf{t}^{\bullet}_2) \Rightarrow & [\text{ functionality of } g^{\bullet}] \\
g^{\bullet} \circ \rho \circ f^{\bullet}(\mathbf{t}^{\bullet}_1) = g^{\bullet}\rho \circ f^{\bullet}(\mathbf{t}^{\bullet}_2) \Rightarrow & [\text{ definition of } \rho^{\#}] \\
\rho^{\#}(\mathbf{t}^{\bullet}_1) = \rho^{\#}(\mathbf{t}^{\bullet}_2), &
\end{array}$$

so that $\mathsf{W}\!\downarrow{\models}ANI^{\rho^{\#}}_{\phi^{\#}}$.

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

Let $\pi_T \in ANI^{\rho^\#}_{\phi^\#}$, we show that $\pi_T \subseteq \tilde{\tau}(\pi_S)$ for some $\pi_S \in ANI^{\rho}_{\phi}$.
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 $\pi_S \in ANI^{\rho}_{\phi}$. Let $s_1, s_2 \in \pi_S$,

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

that shows $\pi_S \in ANI^{\rho}_{\phi}$ and concludes the proof.

□

Proof of Theorem 5.2. Assume $W \models ANI^{\rho}_{\phi}$ and $W\downarrow \rightsquigarrow t_1, t_2$ with $\phi^\#(t_1^\circ) = \phi^\#(t_2^\circ)$. We have to show that $\rho^\#(t_1^\bullet) = \rho^\#(t_1^\bullet)$, for an arbitrary $\rho^\#$ that satisfies the condition

$$H \equiv \forall s\ t.\ s^\bullet \overset{\bullet}{\sim} t^\bullet \Rightarrow \rho^\#(\tilde{\tau}^\bullet(\rho(s^\bullet))) = \rho^\#(t^\bullet).$$

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. Morevoer notice that by functionality and totality, of $\overset{\circ}{\sim}$, $f^\circ(t_1^\circ) = \{s_1^\circ\}$ and $f^\circ(t_2^\circ) = \{s_2^\circ\}$.

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

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

□

Proof of Theorem 5.3. Assume $W \models ANI^{\rho^\#}_{\phi^\#}$ and $W\downarrow \rightsquigarrow t_1, t_2$ with $\phi(t_1^\circ) = \phi(t_2^\circ)$ and ϕ satisfying the condition $H \equiv \forall s\ t.\ s^\circ \overset{\circ}{\sim} t^\circ \Rightarrow \phi(t^\circ) = \phi(\tilde{\tau}^\circ(s^\circ))$. We have to show that $\rho(t_1^\bullet) = \rho(t_2^\bullet)$. 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 $\tilde{\sigma}^\bullet$ is injective. Morevoer notice that by functionality and totality, of $\overset{\bullet}{\sim}$, $\tilde{\tau}^\bullet(s_1^\bullet) = \{t_1^\bullet\}$ and $\tilde{\tau}^\bullet(s_2^\bullet) = \{t_2^\bullet\}$.

$$\begin{aligned}
&\phi(t_1^\circ) = \phi(t_2^\circ) \Rightarrow && [\text{ by } H]\\
&\phi(\tilde{\tau}^\circ(s_1^\circ)) = \phi(\tilde{\tau}^\circ(s_2^\circ)) \Rightarrow && [\text{ functionality of } \tilde{\sigma}^\circ]
\end{aligned}$$

$$\begin{aligned}
&\tilde{\sigma}^{\circ}(\phi(\tilde{\tau}^{\circ}(\mathsf{s}^{\circ}_{1}))) = \tilde{\sigma}^{\circ}(\phi(\tilde{\tau}^{\circ}(\mathsf{s}^{\circ}_{1}))) \Rightarrow && [\text{ definition of } \phi^{\#}]\\
&\phi^{\#}(\mathsf{s}^{\circ}_{1}) = \phi^{\#}(\mathsf{s}^{\circ}_{2}) \Rightarrow && [\mathsf{W} \models ANI^{\phi^{\#}}_{\phi^{\#}}]\\
&\rho^{\#}(\mathsf{s}^{\bullet}_{1}) = \rho^{\#}(\mathsf{s}^{\bullet}_{2}) \Rightarrow && [\text{ by definition of } \rho^{\#}]\\
&\tilde{\sigma}^{\bullet}(\rho(\tilde{\tau}^{\bullet}(\mathsf{s}^{\bullet}_{1}))) = \tilde{\sigma}^{\bullet}(\rho(\tilde{\tau}^{\bullet}(\mathsf{s}^{\bullet}_{2}))) \Rightarrow && [\text{ injectivity of } \tilde{\sigma}^{\bullet}]\\
&\rho(\tilde{\tau}^{\bullet}(\mathsf{s}^{\bullet}_{1})) = \rho(\tilde{\tau}^{\bullet}(\mathsf{s}^{\bullet}_{2})) \Rightarrow && [\tilde{\tau}^{\bullet}(\mathsf{s}^{\bullet}_{\mathsf{i}}) = \{\mathsf{t}^{\bullet}_{\mathsf{i}}\}\ i = 1, 2]\\
&\rho(\mathsf{t}^{\bullet}_{1}) = \rho(\mathsf{t}^{\bullet}_{2})
\end{aligned}$$

so that $\mathsf{W}\downarrow \models ANI^{\rho}_{\phi}$.

□

Proof of Theorem 6.1 (🐓). Theorems rel_RTC_τRTP and rel_RTC_σRTP in `RobustTraceCriterion.v`. □

Proof of Theorem 6.3 (🐓). Theorems tilde_RSC_σRSP and tilde_RSC_Cl_τRTP in `RobustSafetyCriterion.v`. □

Proof of Theorem 6.5 (🐓). Lemmas σRHP_rel_RHC and rel_RHC_σRHP and Theorem rel_RHC_τRHP in `RobustHyperCriterion.v`. □

Proof of Theorem 7.1 (🐓). (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* [? ? ? ?], aided by a few straightforward lemmas and where the case of function calls is guaranteed to terminate by the language. □

Proof of Theorem 3.6 (🐓). Theorems tilde_SC_σSP and tilde_SC_Cl_τTP in `SafetyCriterion.v`. □

Proof of Theorem 3.7. For the implication from left to right, assume $\mathsf{W} \models H$. By CC$^{=}$ have $\mathbf{beh}(\mathsf{W}\downarrow) = \mathsf{beh}(\mathsf{W})$, so that $\mathsf{W}\downarrow \models H$ as well.
For the implication from right to left, instantiate HP with the hyperproperty $\{\mathsf{beh}(\mathsf{W})\}$, for a given W, and deduce that $\mathsf{W}\downarrow \models \{\mathsf{beh}(\mathsf{W})\}$ i.e., $\mathbf{beh}(\mathsf{W}\downarrow) = \mathsf{beh}(\mathsf{W})$. □

Proof of Theorem 3.8 (🐓). Theorems rel_HC_τHP, rel_HC_σHP and σHP_rel_HC in `HyperCriterion.v`. □