Secure Compilation as Hyperproperties Preservation

Marco Patrignani   Deepak Garg

Technical report MPI-SWS-2017-002

June 2017

Abstract

The area of secure compilation aims to design compilers which produce hardened code that can withstand attacks from low-level co-linked components. So far, there is no formal correctness criterion for secure compilers that comes with a clear understanding of what security properties the criterion actually provides. Ideally, we would like a criterion that, if fulfilled by a compiler, guarantees that large classes of security properties of source language programs continue to hold in the compiled program, even as the compiled program is run against adversaries with low-level attack capabilities. This paper provides such a novel correctness criterion for secure compilers, called *trace-preserving compilation* (**TPC**). We show that **TPC** preserves a large class of security properties, namely all safety hyperproperties. Further, we show that **TPC** preserves more properties than full abstraction, the de-facto criterion used for secure compilation. Then, we show that several fully abstract compilers described in literature satisfy an additional, common property, which implies that they also satisfy **TPC**. As an illustration, we prove that a fully abstract compiler from a typed source language to an untyped target language satisfies **TPC**.

This paper uses colours to distinguish elements of different languages. For a good experience, please print/view it in colour.

1 Introduction

Many high-level languages offer security features to programmers in the form of type systems, encapsulation primitives and so forth. Programs written in these high-level languages are ultimately translated into executable code in a low-level, target language by a compiler. Unfortunately, most target languages do not offer the same security features as high-level source languages, so target-level programs are subject to attacks such as control flow hijacking as well as reading/writing of private data or even code. One way to prevent these attacks is to use a compiler that produces target-level programs that are as secure as their source-level counterparts. Such compilers are called *secure compilers*.

Researchers have investigated secure compilation predominantly in the form of fully abstract compilation (or analogous notions) [1, 40, 7, 8, 17, 25, 5, 28, 21, 20, 42, 33, 2, 3, 4, 32, 38, 30], which means that source-level program equivalence is preserved and reflected by compilation or, in other words, two source-level programs are equivalent iff their compilations are equivalent. Fully abstract compilation is a useful extensional soundness criterion for secure compilers, as it ensures the absence of target-level attacks like control flow hijacks *a priori*. However the specific security properties preserved by a fully abstract compiler depend on the definition of program equivalence in the source and target languages. In particular, to preserve different security properties, the definitions of equivalence must be changed appropriately. This variable definition of full abstraction does not yield a *criterion* for compiler security. As a result, many existing work on secure compilation [40, 5, 25, 7, 8, 21, 28, 42] uses a *single, standard* notion of full abstraction obtained by defining program equivalence as contextual equivalence in all possible contexts. However, it is unclear what security properties are preserved by this criterion and, as we show later, obvious security properties are *not* preserved by it (Section 2).

Motivated by this, we ask whether we can find a different criterion for soundness of secure compilers that is guaranteed to preserve a large class of security properties. As a first step in this direction, we present *trace preserving compilation* (**TPC**), an *intensional* criterion for compiler correctness that we show preserves an entire class of security properties, namely all hypersafety properties [19]. **TPC** is based on the notion of trace semantics. Intuitively, a trace-preserving compiler generates code modules that (i) preserve the behaviour of their source-level counterparts when the low-level environment provides valid inputs and (ii) correctly identify and recover from invalid inputs. Invalid inputs are those that have no source-level counterpart (e.g., if booleans are encoded as the integers 0 and 1 by a compiler, then 2 would be an invalid boolean input in the target). Condition (i) implies what is often called *correct compilation*—that the target preserves source behaviour when all interacting components have been compiled using the same (or an equivalent) compiler. Condition (ii) ensures that compiled code detects and responds appropriately to target-level attacks. (We provide a more detailed overview of these notions in Section 2).

Technically, we identify two different strategies for responding to invalid inputs, thus obtaining two slightly different characterizations of **TPC** (Section 3). After defining **TPC**, we prove that it preserves all hypersafety properties (Section 4). In prior work, hypersafety properties have been shown to capture many security-relevant properties including all safety properties as well as information flow properties like noninterference [19]. Hence, showing that **TPC** preserves all hypersafety properties implies that it also preserves these specific properties.

Next, we study the relationship between **TPC** and fully abstract compilation. We show that **TPC** is stronger than the standard notion of fully abstract compilation (Section 5) under *injectivity*, a specific condition on translation of input and output symbols that full abstraction necessarily requires. We further show that under another assumption, which we call fail-safe behaviour or *FSB* (compiled code modules immediately terminate on invalid inputs), correct compilation is equivalent to (a form of) **TPC** (Section 6). As all existing fully-abstract compilers are also correct, we use *FSB* as a means to show that existing fully abstract compilers really achieve **TPC**.

The formal setting in which we study **TPC** is deterministic reactive programs. This is the minimal interesting setting in which one can examine trace-based notions such as hyperproperties. Our formal model of reactive programs abstracts over the code of modules, retaining only their I/O behavior. This suffices for defining **TPC**. However, in writing a compiler, one must be concerned with the code. To bridge this gap, we show by example in Section 6 how our definition applies to a concrete language (a typed lambda-calculus) and a concrete compiler for it. Specifically, we state *FSB* in terms of contextual equivalence and show that the fully abstract compiler of Devriese *et al.* [21] satisfies *FSB* and is, therefore, **TPC**. This implies that the compiler preserves all hypersafety properties. We believe this observation also applies to other existing secure compilers [7, 8, 28, 5, 40, 30, 25, 42].

To summarize, the contributions of this paper are:

- a new intensional soundness criterion for secure compilation (**TPC**);
- a proof that **TPC** preserves all hypersafety properties;
- the relation between **TPC** and fully abstract compilation, the current standard for secure compiler correctness, and a proof that **TPC** is stronger;
- a characterization of a property that existing fully abstract compilers satisfy, which implies that they also satisfy **TPC**, hence showing that **TPC** already exists in current secure compilers.

For space constraints, full proofs and additional discussion can be found in the appendix.

2 Informal Overview

This section provides an overview of the programming model and compilers we consider (Section 2.1 and Section 2.2 respectively). Then it defines compiler properties (Section 2.3) such as correctness and full abstraction. We then present shortcomings of compiler full abstraction to motivate the need for a new compiler soundness criterion (Section 2.4). Finally, we discuss the contributions of this paper—the new soundness criterion for secure compilation (Section 2.5).

Colour conventions: We use a **blue, bold** font for *source* elements, red, sans-serif one for *target* elements and *black* for elements common to both languages to avoid repeating the same definition twice. Thus, $\mathbf{C}$ is a source-level program, C is a target-level one and C is generic notation for either a source-level or a target-level program.

2.1 Reactive Programs

We study the secure compilation of *deterministic reactive programs*. A reactive program contains some internal state, which is not directly observable and reacts to a stream of *inputs* from the environment by producing a stream of observable *outputs*. After each input, the program may update its internal state, allowing all past inputs to influence an output. By definition, a reactive program is really a *component* of a larger program that provides it inputs, i.e., it is a *partial program* (we use the terms components and programs to refer to the same notion).

Definition 1 (Reactive language)**.** A reactive language is a quintuple (I, O, P, ρ). I, O are sets of input and output actions. P is a set of components (all sets we consider are finite or countably infinite). $\rho : I \times P \to O \times P$ is a transition function that represents the language semantics. We overload the notation and use P for program states too. Elements of I, O and P are written $\alpha?$, $\alpha!$ and C, respectively. When component C is given input $\alpha?$, it produces the output $\alpha!$ and advances internally to the state C' if $\rho(\alpha?, C) = (\alpha!, C')$. Termination (as well as divergence) are special outputs after which the component keeps responding only with the same action, so it *stutters*.

A reactive program includes mutable, unobservable internal state as well as code. The code is left abstract but we often use concrete syntax in examples and explanations. Implicitly, the considered programs are *input total*, i.e., they react to all possible inputs. We use the adjective "initial" with a program to indicate the situation prior to any interaction with the environment.

Definition 2 (Traces)**.** A trace, written $\bar{\alpha}$, is an infinite sequence of alternating input-output actions, so $\bar{\alpha} \equiv \alpha_1?, \alpha_1!, \alpha_2?, \alpha_2!, \ldots$ where $\equiv$ denotes syntactic equivalence. All actions are taken from the alphabet $A^\alpha = I \cup O$. Whenever we write α, we implicitly mean $\alpha \in A^\alpha$.

A trace $\alpha_1?\alpha_1! \cdots$ *is in the behaviours* of an initial program C^0 when there is a sequence of states $C_1, \ldots, C_n, \ldots$ such that for each $j \geq 1$, $\rho(\alpha_j?, C_{j-1}) = (\alpha_j!, C_j)$. The set of all traces of C^0 is written $TR(C^0)$.

In general, two programs are said to be *contextually equivalent* when they cannot be distinguished by any context. In our reactive setting, contextual equivalence coincides with trace equivalence.

Definition 3 (Trace equivalence)**.** Two programs are trace equivalent, written $C_1 \overset{\mathrm{T}}{=} C_2$, if their trace semantics coincide. $C_1 \overset{\mathrm{T}}{=} C_2 \overset{\mathsf{def}}{=} TR(C_1) = TR(C_2)$.

We now present an example of a trivial reactive language that we use later.

Example 1 (A reactive language for booleans)**.** Consider a source language $\mathcal{S}$ that only includes terminating programs that compute the boolean identity function. Internally, these programs can do arbitrary computation, but they take a boolean as input and produce the same boolean as output. We omit the full syntax and semantics of *internal* reductions, which can be though of as a typed lambda calculus. Intuitively, input actions $\mathbf{i_{id}}$ can be thought of as function calls, while output ones $\mathbf{o_{id}}$ can be seen as returns.

$$\begin{array}{lr} \text{inputs} & \mathbf{i_{id}} = \{\ \mathbf{id(true)}?,\ \mathbf{id(false)}?\} \\ \text{outputs} & \mathbf{o_{id}} = \{\ \mathbf{ret(true)}!,\ \mathbf{ret(false)}!\} \\ \text{programs} & \mathbf{id} \overset{\mathsf{def}}{=} \lambda\mathbf{x.x},\ \mathbf{id_{not}} \overset{\mathsf{def}}{=} \lambda\mathbf{x.not\ (not\ x)}, \cdots \end{array}$$

Any infinite concatenation of the two trace fragments below describe the possible behaviour of *any* program in $\mathcal{S}$.

$$\boldsymbol{\alpha}_\mathbf{t} \overset{\mathsf{def}}{=} \mathbf{id(true)}? \cdot \mathbf{ret(true)}!$$
$$\boldsymbol{\alpha}_\mathbf{f} \overset{\mathsf{def}}{=} \mathbf{id(false)}? \cdot \mathbf{ret(false)}!$$

Since the traces of all programs in $\mathcal{S}$ are the same, any two programs in $\mathcal{S}$ are trace-equivalent. ⊡

2.2 Compilers

A compiler is a tool that (among other things) transforms initial programs of a source language to initial programs of a target language, relative to a coding of source inputs and outputs in the target. Let $\mathcal{S} = (\mathbf{I}, \mathbf{O}, \mathbf{P}, \boldsymbol{\rho})$ and $\mathcal{T} = (\mathsf{I}, \mathsf{O}, \mathsf{P}, \rho)$ be a source and a target language, respectively.

Definition 4 (Compiler)**.** A compiler from $\mathcal{S}$ to $\mathcal{T}$ is a triple $(\approx_I, \approx_O, [\![\cdot]\!]^{\mathcal{S}}_{\mathcal{T}})$, where $\approx_I$ and $\approx_O$ are relations on $\mathbf{I} \times \mathsf{I}$ and $\mathbf{O} \times \mathsf{O}$ that represent coding of inputs and outputs respectively, and $[\![\cdot]\!]^{\mathcal{S}}_{\mathcal{T}} : \mathbf{P} \to \mathsf{P}$ is a function that translates source initial components to target initial ones. We assume that $\approx_I$ and $\approx_O$ satisfy the following two conditions (stated here only for $\approx_I$ for brevity):
(Totality) For every $\alpha? \in \mathbf{I}$, there exists $\alpha? \in \mathsf{I}$ such that $\alpha? \approx_I \alpha?$.
(Functionality) $\boldsymbol{\alpha_1}? \approx_I \alpha?$ and $\boldsymbol{\alpha_2}? \approx_I \alpha?$ imply $\boldsymbol{\alpha_1}? = \boldsymbol{\alpha_2}?$

Relations $\approx_I$ and $\approx_O$ specify how inputs and outputs are coded by the compiler. For instance, if a compiler maps the input **true** to the input 1, then we would have $\mathbf{true} \approx_I 1$. Totality is essential since a compiler should consider all source behaviour. Functionality is not necessary for compilers in general, but in the context of preserving security properties, it is essential to avoid conflating (through compilation) distinct source symbols that a property of interest treats differently. For example, in information flow security, relating a public and a private source action to the same target action would make it impossible to talk about the preservation of a property like noninterference.

Throughout this paper, we write $\approx$ in place of both $\approx_I$ and $\approx_O$ and often refer to a compiler as just the function $[\![\cdot]\!]^{\mathcal{S}}_{\mathcal{T}}$, assuming implicitly that $\approx$ is given. $\approx$ is lifted to traces point-wise (Rule Relate-trace).

$$\frac{\boldsymbol{\alpha_1} \approx \alpha_1 \qquad \boldsymbol{\alpha_2}, \cdots \approx \alpha_2, \cdots}{\boldsymbol{\alpha_1}, \boldsymbol{\alpha_2}, \cdots \approx \alpha_1, \alpha_2, \cdots}\ \text{(Relate-trace)}$$

2.3 Compiler Properties

This section presents two compiler properties, correctness and full abstraction, that are often used together as a criteria for the soundness of a secure compiler. Correctness (Definition 5) states that the translation of programs agrees with the translation of inputs and outputs, i.e., the compilation preserves and reflects source program behaviour. Define the set of all correct compilers as CC.

Definition 5 (Compiler correctness)**.** A compiler $[\![\cdot]\!]^{\mathcal{S}}_{\mathcal{T}}$ is correct, denoted $[\![\cdot]\!]^{\mathcal{S}}_{\mathcal{T}} \in CC$, if $\forall \mathbf{C}, \bar{\boldsymbol{\alpha}}, \bar{\alpha}.\ \bar{\boldsymbol{\alpha}} \in \mathbf{TR}(\mathbf{C})$ and $\bar{\boldsymbol{\alpha}} \approx \bar{\alpha}$ imply $\bar{\alpha} \in \mathsf{TR}([\![\mathbf{C}]\!]^{\mathcal{S}}_{\mathcal{T}})$.

Compiler full abstraction is the most-widely used soundness criterion for secure compilation. It states that the compiler preserves and reflects some notion of program equivalence. The general idea behind full abstraction is that the abilities of the context (the attacker) often differ between the source and the target. For instance, in a target language that is assembly, the context may be able to access private fields of an object directly through load/store instructions, but this access may be prohibited to source-level contexts by the source semantics. A fully abstract compiler can rule out such attacks by ensuring (often through dynamic checks) that the power of an attacker interacting with the compiled program in the target language is limited to attacks that could also be performed by some source language attacker interacting with the source program.

Nonetheless, the specific security properties preserved by a secure compiler depend on the chosen notion of program equivalence. In the secure compilation literature, the most commonly chosen notion of program equivalence is contextual equivalence (indistinguishability by any context in the language), which, as noted before, coincides with trace equivalence in our setting. This corresponds to the following definition of full abstraction. We re-emphasize that this is just one possible definition of full abstraction (the most commonly used), based on the most commonly used notion of program equivalence.

Definition 6 (Full abstraction)**.** A compiler $[\![\cdot]\!]^{\mathcal{S}}_{\mathcal{T}}$ is fully abstract, denoted as $[\![\cdot]\!]^{\mathcal{S}}_{\mathcal{T}} \in FA$, if $\forall \mathbf{C}, \mathbf{C}'$. we have that $\mathbf{C} \overset{\mathrm{T}}{=} \mathbf{C}'$ if and only if $[\![\mathbf{C}]\!]^{\mathcal{S}}_{\mathcal{T}} \overset{\mathrm{T}}{=} [\![\mathbf{C}']\!]^{\mathcal{S}}_{\mathcal{T}}$.

2.4 Shortcomings of Full Abstraction for Security

Most existing work on secure compilation proves (or assumes) compiler correctness and proves compiler full abstraction in the sense defined above. We now show that, in fact, some intuitive and interesting security properties are *not* necessarily preserved by such a compiler. This justifies the need for a new soundness criterion for secure compilation. The need for new soundness criteria has also pointed out by other recent work [42, 30, 43].

Example 2 (Safety violation)**.** Consider a source language whose only data type is booleans that only admits the constant function that always returns **true**. Consider a target language $\mathcal{T}$ (called $\lambda^{\mathbb{N}}$ in the remainder of the paper) whose programs perform operations on natural numbers, so they input numbers and output numbers. Consider a trivial compiler $[\![\cdot]\!]^{\mathcal{S}}_{\mathcal{T}}$ from $\mathcal{S}$ to $\mathcal{T}$ that maps any source program to the target program λx. if x $<$ 2 then 1 else 0.

Under the coding **true** $\approx$ 1 and **false** $\approx$ 0, this compiler is both correct and fully abstract (trivially). However, this compiler does not preserve even trivial properties like "never output false". In the source, this property cannot be violated (since the only allowed functions always output **true**). In the target, the property would naturally translate to "never output 0" but on input 2, compiled programs output 0 and violate the property. ⊡

One may argue that there is, in fact, a gap in this argument since we have not formally specified how to translate a source property to the target language and have relied on an intuitive translation. Indeed, there is no single canonical translation of properties in literature, and this problem has been identified by Abadi over a decade ago [1]. However, for a *safety* property like the one above, where the goal is for the program to not reach an unsafe state (or produce an unsafe output), a natural translation would rule out the translations of outputs that the source property rules out.

To summarize, this example identifies two problems:

1. It is unclear what it means to preserve a source-language property in the target language as the two languages can be different (this subject is further developed in Section 4.2);

2. Standard full abstraction and compiler correctness do not preserve all safety properties under an intuitive translation of properties.

The novel secure compilation criterion we propose preserves all safety properties under a translation that we prove to preserve the intuitive meaning of safety properties (as described in Section 4.2.3).

Example 3 (Confidentiality violation)**.** Consider the source language of Example 2 but with a simple addition: these programs now store a boolean secret in their internal state. All programs of this language always returns **true** except on the 10th input, where they output the boolean secret. Their traces can therefore be as in Figure 1 (the indices t and f indicate the internally stored secret).

Consider the following declassification property: "Do not output the secret until the 10th input". All source programs satisfy this property.

Consider $\lambda^{\mathbb{N}}$, the target language of Example 2 that inputs and outputs natural numbers, and the same coding $\approx$. Consider a compiler that translates source programs to behave exactly as in the source if the input is 1 or 0 (the encodings of **true** and **false**), but to output the secret *immediately* if the input is any number greater than 1. A subset of the semantics of compiled components is also presented in Figure 1.

This compiler is correct, because correctness is concerned only with target traces that are translated from the source, and compiled components have traces $\alpha_{\mathtt{t}}$ and $\alpha_{\mathtt{f}}$ (as well as elided ones) that derive from $\boldsymbol{\alpha}_{\mathbf{t}}$ and $\boldsymbol{\alpha}_{\mathbf{f}}$. The compiler

$$
\begin{aligned}
\boldsymbol{\alpha}_{\mathsf{t}} &= \overbrace{\mathbf{id}(\mathbf{true})? \cdot \mathbf{ret}(\mathbf{true})!}^{\textit{nine times}} \cdot \mathbf{id}(\mathbf{true})? \cdot \overset{\text{secret}}{\mathbf{ret}(\mathbf{true})!} \cdots \\
\boldsymbol{\alpha}_{\mathsf{f}} &= \overbrace{\mathbf{id}(\mathbf{true})? \cdot \mathbf{ret}(\mathbf{true})!}^{\textit{nine times}} \cdot \mathbf{id}(\mathbf{true})? \cdot \mathbf{ret}(\mathbf{false})! \cdots \\
\boldsymbol{\alpha}'_{\mathsf{t}} &= \overbrace{\mathbf{id}(\mathbf{false})? \cdot \mathbf{ret}(\mathbf{true})!}^{\textit{nine times}} \cdot \mathbf{id}(\mathbf{false})? \cdot \mathbf{ret}(\mathbf{true})! \cdots \\
\boldsymbol{\alpha}'_{\mathsf{f}} &= \overbrace{\mathbf{id}(\mathbf{false})? \cdot \mathbf{ret}(\mathbf{true})!}^{\textit{nine times}} \cdot \mathbf{id}(\mathbf{false})? \cdot \mathbf{ret}(\mathbf{false})! \cdots
\end{aligned}
$$

$$
\begin{aligned}
\alpha_{\mathsf{t}} &= \overbrace{\mathsf{id}(1)? \cdot \mathsf{ret}(1)!}^{\textit{nine times}} \cdot \mathsf{id}(1)? \cdot \overset{\text{secret}}{\mathsf{ret}(1)!} \cdots \\
\alpha_{\mathsf{f}} &= \overbrace{\mathsf{id}(1)? \cdot \mathsf{ret}(1)!}^{\textit{nine times}} \cdot \mathsf{id}(1)? \cdot \mathsf{ret}(0)! \cdots \\
\alpha_{\mathsf{u}} &= \overbrace{\mathsf{id}(1)? \cdot \mathsf{ret}(1)!}^{\textit{less than nine times}} \cdot \mathsf{id}(2)? \cdot \mathsf{ret}(1)! \cdots \\
\alpha'_{\mathsf{u}} &= \overbrace{\mathsf{id}(1)? \cdot \mathsf{ret}(1)!}^{\textit{less than nine times}} \cdot \mathsf{id}(2)? \cdot \mathsf{ret}(0)! \cdots
\end{aligned}
$$

Figure 1: Source and target traces for Example 3.

is also fully abstract: source programs with the same trace semantics have the same trace semantics at the target as well.

However, again, the compiled programs do not satisfy the intended declassification property: they can be caused to leak the secret at any time, even before the 10th input, as in α_{u} and α'_{u} by providing 2 as the input. ⊡

A bit of analysis shows the precise shortcoming of compiler correctness and full abstraction as a joint soundness criteria in these examples and for secure compilation in general. Call a target input $\alpha?$ *invalid* if it does not code a source input, i.e., if there is no $\boldsymbol{\alpha}?$ such that $\boldsymbol{\alpha}? \approx \alpha?$. Compiler correctness does not handle these inputs since it states that a compiled program should behave exactly like the source program while the (target) inputs are *valid*. However, once an invalid input is received by a compiled program, compiler correctness does not constrain the behaviour of the program further. It is this lack of constraint that Example 2 exploits. Furthermore, for a pair of *distinguishable* source programs, full abstraction says nothing if the compiler is correct. Consequently, two distinguishable source programs are allowed to differ in an *arbitrary* manner after an invalid input is received in the target, while the property of interest may care *how* the programs differ. Example 3 exploits this freedom.

(Some readers may argue that we should not call a compiler correct if we do not consider all possible inputs that the compiled code can receive. We argue otherwise: compiler correctness is always defined for programs that interact with target-level programs that also have source-level counterparts—often they are obtained via the same compiler—because that is what is expected in the absence of an adversary.)

It should be clear that the problem here is the freedom of behaviour on invalid inputs. The novel compiler security criterion we propose (**TPC**) curtails this freedom by defining precisely how the program should behave on invalid inputs.

Remark A viable criticism of our analysis of Examples 2 and 3 is that one could change the notion of the source and target program equivalence in the definition of full abstraction to capture the required properties precisely. In fact, early work on full abstraction for secure compilation [4] kept the choice of the program equivalence relation open. However, note that a flexible definition of full abstraction does not lend itself to a viable criterion for compiler design. When the compiler is written, one may not know what properties would be of interest for programs that will be compiled later, so what notion of full abstraction should the compiler adhere to? In contrast, what we propose is a *fixed* criterion for compiler security that preserves classes of security properties.

2.5 Trace-Preserving Compilation (TPC), Informally

Informally, a compiler $[\![\cdot]\!]^{\mathcal{S}}_{\mathcal{T}}$ from $\mathcal{S}$ to $\mathcal{T}$ is trace-preserving (Definition 1) if it produces components $[\![\mathbf{C}]\!]^{\mathcal{S}}_{\mathcal{T}}$ whose traces are either source-level traces ($\mathbf{TR}(\mathbf{C})$) or invalid traces ($\mathsf{B}_{\mathbf{C}}$).

Informal definition 1 (Trace-preserving compiler, informally)**.** $\forall \mathbf{C} \in \mathcal{S}.\ \mathsf{TR}([\![\mathbf{C}]\!]^{\mathcal{S}}_{\mathcal{T}}) = \mathbf{TR}(\mathbf{C}) \cup \mathsf{B}_{\mathbf{C}}$.

The first part of the union in Definition 1 states that traces of a compiled component $\mathbf{C}$ must include *all* the source-level traces of $\mathbf{C}$ (i.e., the "valid traces"). This ensures that a **TPC** compiler is correct. The second part of the union, $\mathsf{B}_{\mathbf{C}}$, contains only traces that contain at least one invalid input (inputs that are not related to anything in the source) and specifies how the compiled code must react to such inputs. Specifically, we require that the output in response to an invalid input be *fresh* and *opaque*. Fresh means that the output must not be related to a source symbol (to prevent outputting of symbols that are forbidden by a source safety property of interest, as in Example 2), while opaque means that the output must not depend on any hidden internal state (to prevent information leaks, as in Example 3). We denote such a fresh and opaque output with a $\surd$.

Example 4 (Invalid traces)**.** Consider the following trace for the source language of Example 1, the target language $\lambda^{\mathbb{N}}$ from Example 2 and the same coding $\approx$ of Example 2. Let $\surd$ be any output in the target that is not related under $\approx$ to any source symbol.

$$\begin{aligned}
\boldsymbol{\alpha}_{\mathbf{valid}} &= \mathbf{id(false)?\ ret\ false!}\ \cdots\\
\boldsymbol{\alpha}_{\mathbf{invalid}} &= \mathbf{id(3)?\ ret\ false!}\ \cdots\\
\alpha_{\mathsf{tick}} &= \mathsf{id}\ (3)?\ \surd\ \cdots
\end{aligned}$$

$\boldsymbol{\alpha}_{\mathbf{valid}}$ is a valid source trace, while $\boldsymbol{\alpha}_{\mathbf{invalid}}$ is a trace that cannot exist in the source. Due to the definition of $\approx$, α_{tick} is a good example of an invalid target trace (as we mean them), as no trace in the source relates to it and it reveals no information about the program's internal state. $\boxdot$

The idea to respond in a fresh and opaque way to ill-formed inputs is not novel. Existing fully abstract secure compilers already react to invalid inputs in such a way [40, 42, 25, 5, 28, 32]. Our contribution, instead, is in formalizing this idea and in establishing formally what it means in terms of preservation of classes of hyperproperties. All the above-mentioned compilers generally have a single way of reacting to an invalid input: they halt the machine (e.g., they reduce to a stuck term *wrong* in the target's semantics). However, one can envisage different implementation strategies for such a response. For example, the invalid input can be disregarded and the program can continue as if nothing happened. Understanding the security relevance (in terms of the preservation of properties) of the possible alternative implementations of $\surd$ is another contribution of this paper.

Concretely, we identify two different ways to respond to invalid inputs:

1. *halting* the component forever;

2. *disregarding* the invalid input.

Item 1 is the strategy used by prior work; it does not return control to the attacker. This strategy also encompasses the case of diverging when an invalid input is detected.

Item 2 covers a different scenario, where the availability of the component is crucial and therefore it must not stop responding in the future. A number of real world applications fall in this scenario, most predominantly servers, which need to continue running even if malformed (possibly malicious) input is received. A possible implementation of this scenario could be that a trusted kernel is notified on invalid input and the kernel resets the compiled component to a known good state. These two ways of responding to invalid inputs yield two different variants of **TPC**, which we call the "halting" and "disregarding" variants.

Some readers may then wonder about specific ways to implement **TPC**. As all inputs need to be checked to understand their validity or not, a way to obtain **TPC** is to correctly compile code and then wrap it with dynamic checks that enforce **TPC**. This strategy, albeit costly, is also how many fully-abstract compilers operate, introducing performance overhead in exchange for security. Means to reduce the overhead come in the form of security architectures such as protected modules architectures [36], the pump machine [22] or capability machines [48]. We leave the investigation of **TPC** compilers for these architectures for future work.

Having defined the formal setting and the intuition behind the contribution of this work, we now define **TPC** formally.

3 Trace-Preserving Compilation

This section defines **TPC** in both its halting and disregarding variants (Definition 7 and Definition 8, respectively).

We introduce some notation used in the remainder of this paper.

Notation 1 (Notation for traces and other formal details)**.**

- given a trace $\bar{\alpha} = \alpha_1?, \alpha_1!, \alpha_2?, \alpha_2!, \ldots$, define functions $\bar{\alpha}|_I$ and $\bar{\alpha}|_O$ to project its inputs and outputs as follows: $\bar{\alpha}|_I = \alpha_1?, \alpha_2?, \ldots$ and $\bar{\alpha}|_O = \alpha_1!, \alpha_2!, \ldots$.
- denote a set of elements of type t as $\widehat{t}$ or $\{t\}$.
- denote the cardinality of a set $\widehat{t}$ as $\|\widehat{t}\|$.
- denote a set of traces as $\widehat{\bar{\alpha}}$.
- denote a set of sets of traces as $\mathbb{T}$ (so it should be $\widehat{\widehat{\bar{t}}}$).
- indicate finite traces prefixes (sometimes called just traces or finite traces with some abuse of terminology) using the metavariable $\bar{m}$.
- $\bar{m} \leq \bar{\alpha}'$ means that $\bar{m}$ is a prefix of $\bar{\alpha}'$, so $\bar{\alpha}' \equiv \bar{m}\bar{\alpha}''$ for some $\bar{\alpha}''$.
- lift the prefix notion to sets of traces as follows: $\widehat{\bar{m}} \leq \widehat{\bar{\alpha}}'$ if $\forall \bar{m} \in \widehat{\bar{m}}, \exists \bar{\alpha}' \in \widehat{\bar{\alpha}}'. \bar{m} \leq \bar{\alpha}'$.
- define the set of odd-length prefixes of a set of traces as follows: $\texttt{op}(\widehat{\bar{\alpha}}) = \{\bar{m}\alpha? \mid \bar{m}\alpha? \leq \widehat{\bar{\alpha}}\}$.
- define the observables of a trace as all the even-length, finite prefixes of that trace: $\texttt{obs}(\bar{\alpha}) = \{\bar{m}'\alpha?\alpha! \mid \bar{m}'\alpha?\alpha! \leq \bar{\alpha}\} \cup \{\epsilon\}$
- lift relation $\approx$ to sets, denoted as $\widehat{\bar{\alpha}} \approxeq \widehat{\bar{\boldsymbol{\alpha}}}$, as: $\forall \bar{\alpha} \in \widehat{\bar{\alpha}}. \exists \bar{\boldsymbol{\alpha}} \in \widehat{\bar{\boldsymbol{\alpha}}}. \bar{\alpha} \approx \bar{\boldsymbol{\alpha}}$ and $\forall \bar{\boldsymbol{\alpha}} \in \widehat{\bar{\boldsymbol{\alpha}}}. \exists \bar{\alpha} \in \widehat{\bar{\alpha}}. \bar{\alpha} \approx \bar{\boldsymbol{\alpha}}$.

The first definition of **TPC** this section formalises is TP^H, the halting variant of **TPC**.

Definition 7 (**TPC**, halting)**.** $[\![\cdot]\!]^{\mathcal{S}}_{\mathcal{T}} \in TP^H \overset{\mathsf{def}}{=} \forall \mathbf{C}$

$$\begin{aligned}
\mathsf{TR}([\![\mathbf{C}]\!]^{\mathcal{S}}_{\mathcal{T}}) = {} & \{\bar{\alpha} \mid \exists \bar{\boldsymbol{\alpha}} \in \mathbf{TR}(\mathbf{C}). \bar{\boldsymbol{\alpha}} \approx \bar{\alpha}\} \cup \\
& \{\bar{\mathsf{m}}\alpha?\surd\bar{\alpha}' \mid \exists \bar{\mathbf{m}} \in \texttt{obs}(\mathbf{TR}(\mathbf{C})). \bar{\mathbf{m}} \approx \bar{\mathsf{m}} \\
& \text{ and } \forall \alpha' \in \bar{\alpha}'|_O. \alpha' \equiv \surd \\
& \text{ and } \nexists \bar{\mathbf{m}}\boldsymbol{\alpha}? \in \texttt{op}(\mathbf{TR}(\mathbf{C})). \bar{\mathbf{m}}\boldsymbol{\alpha}? \approx \bar{\mathsf{m}}\alpha?\}
\end{aligned}$$

The first component of the union is the set of valid traces, i.e., those that have a source-level counterpart. We often refer to this set as $\mathsf{G}_{\mathbf{C}}$, so $\mathsf{G}_{\mathbf{C}} = \{\bar{\alpha} \mid \exists \bar{\boldsymbol{\alpha}} \in \mathbf{TR}(\mathbf{C}).\bar{\boldsymbol{\alpha}} \approx \bar{\alpha}\}$. The second component of the union, which we refer to as $\mathsf{B}_{\mathbf{C}}$, is the set of invalid traces, which contain a prefix of a valid trace ($\bar{\mathsf{m}}$) followed by an invalid action ($\alpha?$) which is responded to with $\surd$. From there on ($\bar{\alpha}'$), all outputs must be $\surd$, i.e., the trace stutters on $\surd$, which is a terminating symbol. From the formal language perspective, we have that $\surd \in \mathsf{O}$.

To define the "disregarding" version of **TPC**, which we write TP, we need additional machinery. Two prefixes are up-to-tick equivalent, written $\bar{\mathsf{m}} \curlywedge \bar{\mathsf{m}}'$, if they are the same once they are stripped of all their $\surd$s and of the input actions immediately preceding them. Let $\widehat{\surd} \subseteq \mathsf{O}$ be a set of target output actions that have no source counterparts and let $\surd_1, \surd_2, \ldots$ be any ordered sequence of actions from $\widehat{\surd}$ whose elements need not be distinct.

Definition 8 (**TPC**, disregarding)**.** $[\![\cdot]\!]^{\mathcal{S}}_{\mathcal{T}} \in TP \overset{\mathsf{def}}{=} \forall \mathbf{C}$

$$\begin{aligned}
\mathsf{TR}([\![\mathbf{C}]\!]^{\mathcal{S}}_{\mathcal{T}}) &= \{\bar{\alpha} \mid \mathtt{obs}(\bar{\alpha}) = \bigcup_{n\in\mathbb{N}} \mathtt{int}_{\mathtt{n}}(\mathbf{C})\} \qquad \text{where}\\
\mathtt{int}_0(\mathbf{C}) &= \{\bar{\mathsf{m}} \mid \exists \bar{\mathbf{m}} \in \mathtt{obs}(\mathbf{TR}(\mathbf{C})), \bar{\mathbf{m}} \approx \bar{\mathsf{m}}\}\\
\mathtt{int}_{\mathtt{n+1}}(\mathbf{C}) &= \{\bar{\mathsf{m}} \mid \bar{\mathsf{m}} \equiv \bar{\mathsf{m}}_1 \alpha? \surd_{\mathsf{n+1}} \bar{\mathsf{m}}_2 \text{ and } \bar{\mathsf{m}}_1\bar{\mathsf{m}}_2 \in \mathtt{int}_{\mathtt{n}}(\mathbf{C})\\
&\qquad \text{and } \forall \bar{\mathsf{m}}' \curlywedge \bar{\mathsf{m}}_1, \nexists \bar{\mathbf{m}}\boldsymbol{\alpha}? \in \mathtt{op}(\mathbf{TR}(\mathbf{C})).\\
&\qquad \bar{\mathbf{m}}\boldsymbol{\alpha}? \approx \bar{\mathsf{m}}'\alpha? \text{ and } \forall \alpha! \in \bar{\mathsf{m}}_2|_{O}.\alpha! \notin \widehat{\surd}\}
\end{aligned}$$

To contemplate all possible interleavings of all possible bad actions, we consider all observables of all actions that a compiled components must have. These observables are defined inductively. The base case identifies the same set $\mathsf{G}_{\mathbf{C}}$ as in Definition 7. The inductive case adds one more invalid action with a $\surd$ response in response to the last invalid input on the trace. Intuitively, $\mathtt{int}_0(\cdot)$ yields all traces with a source-level counterpart, while $\mathtt{int}_{\mathtt{n}}(\cdot)$ yields all traces that contain exactly n invalid inputs, to which the compiled program responds with $\surd_1, \ldots, \surd_{\mathsf{n}}$ respectively. $\surd$s must be used monotonically based on the ordering of the sequence because they should convey only information that is already available to the environment after an interaction, i.e., the number of past interactions.

By definition, the halting version of **TPC** implies the disregarding one. To see this, given a $\surd$ in the halting version, one can choose $\widehat{\surd} = \{\surd\}$ and the sequence $[\surd, \surd, \ldots]$ in the disregarding version.

Theorem 1 (Halting implies disregarding)**.** $[\![\cdot]\!]^{\mathcal{S}}_{\mathcal{T}} \in TP^{H} \Rightarrow [\![\cdot]\!]^{\mathcal{S}}_{\mathcal{T}} \in TP$.

Next, we relate **TPC** to refinement: If a compiler is TP, then the behaviours of a source program are contained in the behaviours of the compiled program (up to $\approx$). Let $\underset{\approx}{\subset}$ denote $\subseteq \circ \approx$.

Theorem 2 (Source programs refine their compiled counterparts)**.** $\forall [\![\cdot]\!]^{\mathcal{S}}_{\mathcal{T}} \in TP, \forall \mathbf{C}.\mathbf{TR}(\mathbf{C}) \underset{\approx}{\subset} \mathsf{TR}([\![\mathbf{C}]\!]^{\mathcal{S}}_{\mathcal{T}})$.

The next theorem states that equivalent programs have the same set of invalid traces.

Theorem 3 (Equivalent programs have the same invalid traces)**.** $\forall \mathbf{C_1}, \mathbf{C_2}.\ \mathbf{C_1} \overset{T}{=} \mathbf{C_2} \Rightarrow \mathsf{B}_{\mathbf{C_1}} = \mathsf{B}_{\mathbf{C_2}}$.

4 Trace-Preserving Compilation and Hyperproperty Preservation

This section describes hyperproperties and their subclasses (Section 4.1). Then it proves what two security-relevant subclasses of hyperproperties, namely safety and hypersafety, are preserved by **TPC** (Section 4.2). Finally, it describes some classes of hyperproperties that are not preserved by **TPC** (Section 4.3).

4.1 Hyperproperties

Hyperproperties [19] are a formal representation of predicates on programs, i.e., they are predicates on sets of traces. They capture many security-relevant properties including not just conventional safety and liveness (i.e., predicates on traces), but also properties like non-interference (i.e., predicates on sets of traces).

Denote the set of hyperproperties as **HP**. An element of this set is denoted $\mathbb{P} \in \mathbf{HP}$. So, $\mathbb{P} = \{\widehat{\overline{m}}_i\}_{i\in I}$ where I is countable. A program C has a hyperproperty $\mathbb{P}$ if $TR(C) \in \mathbb{P}$.

The following formalisation is taken from the work of Clarkson and Schneider [19] and adapted to our notion of traces. Denote a sequence of infinite observables with $\bar{\alpha}^\omega$ and of finite observables with $\bar{\alpha}^n$ (for some natural number n). The set of infinite traces is denoted with Φ_{inf} while that of finite traces is denoted with Φ_{fin}. We lift these concepts to the hyperproperty level by introducing *Prop* and *Obs*. Let $\mathcal{P}$ denote the powerset function and $\mathcal{P}^f$ the set of all finite subsets.

Definition 9 (Φ_{inf},Φ_{fin},*Prop* and *Obs*)**.**

$$\begin{aligned}
\bar{\alpha}^\omega &= \{\bar{\alpha} \mid \nexists n \in \mathbb{N}. \bar{\alpha} \equiv \alpha_1, \cdots, \alpha_n\} \\
\bar{\alpha}^n &= \{\bar{\alpha} \mid \bar{\alpha} \equiv \alpha_1, \cdots, \alpha_n \wedge n \in \mathbb{N}\}
\end{aligned}$$

$$\Phi_{inf} \overset{\mathsf{def}}{=} \{\bar{\alpha} \mid \bar{\alpha} \in \bar{\alpha}^\omega\} \qquad \Phi_{fin} \overset{\mathsf{def}}{=} \{\bar{\alpha} \mid \bar{\alpha} \in \bar{\alpha}^n\}$$

$$\mathit{Prop} \overset{\mathsf{def}}{=} \mathcal{P}(\Phi_{inf}) \qquad \mathit{Obs} \overset{\mathsf{def}}{=} \mathcal{P}^f(\Phi_{fin})$$

Two core classes of hyperproperties exist: safety hyperproperties (also called hypersafety) and liveness hyperproperties (also called hyperliveness), which are described below.

Given a property p its equivalent hyperproperty, called its lift, is denoted $\lceil p \rceil$. By definition, $\lceil p \rceil = \mathcal{P}(p)$.

4.1.1 Hypersafety

A hyperproperty is hypersafety if it does not allow bad things to happen. Let **SHP** denote the set of all safety hyperproperties.

Definition 10 (Hypersafety)**.** $\mathbb{P} \in \mathbf{SHP} \overset{\mathsf{def}}{=} \forall \widehat{\widehat{\alpha}} \in \mathit{Prop}.\ \widehat{\widehat{\alpha}} \notin \mathbb{P} \Rightarrow (\exists \widehat{\widehat{m}} \in \mathit{Obs}. \widehat{\widehat{m}} \leq \widehat{\widehat{\alpha}}$ and $(\forall \widehat{\widehat{\alpha'}} \in \mathit{Prop}. \widehat{\widehat{m}} \leq \widehat{\widehat{\alpha'}} \Rightarrow \widehat{\widehat{\alpha'}} \notin \mathbb{P}))$.

Intuitively, for a hyperproperty to be hypersafe, all the sets of traces in it must not contain all prefixes in any of the $\widehat{\widehat{m}}$'s that specify "bad things". The set of all the $\widehat{\widehat{m}}$'s characterizes the hypersafety property. The lift of safety properties is a subset of **SHP** and it is denoted as $\lceil S \rceil$.

Example 5 (**SHP** examples)**.** Examples of **SHP** include termination-insensitive non-interference, observational determinism and all safety properties [19]. ⊡

4.1.2 Hyperliveness

A hyperproperty is hyperlive if it always allows for a good thing to happen (Definition 11). Let **LHP** denote the set of all safety hyperproperties.

Definition 11 (Hyperliveness)**.** $\mathbb{L} \in \mathbf{LHP} \overset{\mathsf{def}}{=} \forall \widehat{\widehat{m}} \in \mathit{Obs}.\ (\exists \widehat{\widehat{\alpha'}} \in \mathit{Prop}. \widehat{\widehat{m}} \leq \widehat{\widehat{\alpha'}} \wedge \widehat{\widehat{\alpha'}} \in \mathbb{L})$.

Every hyperproperty is the intersection of a safety hyperproperty and a liveness hyperproperty.

Theorem 4 (**HP** composition [19])**.** $\forall \mathbb{P} \in \mathbf{HP}.\ \exists \mathbb{S} \in \mathbf{SHP}, \mathbb{L} \in \mathbf{LHP}.\ \mathbb{P} = \mathbb{S} \cap \mathbb{L}$.

4.2 Preserving hypersafety via *TP*

The question that we want to address next is: how can one translate a property from a source language to a target language and preserve it (up to the translation) via compilation? "Meaning preservation" is the trickier part of the question, because the two languages are often so different that this is unclear. In fact, we do not believe that there is a general way to translate arbitrary (hyper)properties. Here, we restrict attention to two subclasses of hyperproperties—safety and hypersafety—which are (a) relevant for many security applications, and (b) easy to treat formally since they can be characterized uniformly: a safety (hypersafety) property can be expressed as a set of bad prefixes (set of set of bad prefixes). For each of the two subclasses, we describe how to translate source properties to the target and show that any **TPC** compiler preserves the properties under this translation. Note that our technical development for hypersafety subsumes that for safety; we present the latter separately only for exposition purposes.

4.2.1 Safety Preservation

We now present our result about safety properties. Informally, a safety property prevents bad things from happening [9]. Formally a safety property S (a set of traces) is characterized as follows:

$$\forall \bar{\alpha}.\ \text{if } \bar{\alpha} \notin S \text{ then } (\exists \overline{m} \leq \bar{\alpha} \text{ and } \forall \bar{\alpha}'.\ \text{if } \overline{m} \leq \bar{\alpha}' \text{ then } \bar{\alpha}' \notin S)$$

A trace ($\bar{\alpha}$) is not valid if it has a "bad" prefix $\overline{m}$ that no valid trace has.

Since the $\overline{m}$ is quantified for all $\bar{\alpha}$, we can redefine a safety property by relying on a set of bad prefixes $\widehat{\overline{m}}$ that is the set obtained by taking all the existentially-quantified $\overline{m}$. A safety property S is thus redefined as follows. Let $\widehat{\overline{m}} :: S$ denote that $\widehat{\overline{m}}$ is the set of all bad prefixes that characterises the safety property S.

$$\text{if } \widehat{\overline{m}} :: S \text{ then } \bar{\alpha} \notin S \text{ iff } \exists \overline{m} \in \widehat{\overline{m}}. \overline{m} \leq \bar{\alpha}$$

In this way we can define a safety property by the set of all possible bad prefixes that a good trace must not have.

Next, we need to translate a safety property from a source to a target language. To do so, we translate the set of bad prefixes from the source to the target language and obtain a set of bad prefixes expressed in the target language. However, there is still a concern: the target language can have more actions that are not expressible in the source—the invalid input actions—and outputs produced in response to them. Ideally, we would like to be conservative with respect to these invalid actions and add any prefix with an invalid input to the set of bad prefixes in the target. This ensures that all good traces in the target safety property relate good traces in the source safety property. This is a safe choice.

However, this ideal translation is unrealisable since the adversarial environment, not the compiled program, provides invalid inputs. Thus, if we call all traces with invalid inputs "bad", then we cannot ever hope to preserve safety properties. To still achieve this, we create a small exception: we admit traces with invalid inputs, if the invalid inputs are immediately succeeded by $\surd$ outputs. This is a reasonable compromise since $\surd$ outputs have no source counterparts (so the source property could not possibly be talking about them), and they reveal no information by definition. This can be generalized to allow the ith invalid input to be followed by $\surd_{\mathsf{i}}$ for some pre-determined sequence $\surd_1, \surd_2, \ldots$ of possibly different $\surd$s.

This idea of translating safety properties is formalised in Definition 12.

Definition 12 (Safety relation)**.** Two sets of prefixes define the same safety property, denoted as $\widehat{\bar{\mathbf{m}}} \overset{\text{SP}}{\approx} \widehat{\bar{\mathsf{m}}}$ if:

$$
\begin{aligned}
\widehat{\bar{\mathsf{m}}} = \{\bar{\mathsf{m}} \mid \exists \bar{\mathbf{m}} \in \widehat{\bar{\mathbf{m}}}. \bar{\mathsf{m}} \approx \bar{\mathbf{m}}\}& \\
\cup \{\bar{\mathsf{m}}\alpha?\alpha! \mid \exists \bar{\mathbf{m}} \in \widehat{\bar{\mathbf{m}}}, \bar{\mathsf{m}}'. \bar{\mathbf{m}} \approx \bar{\mathsf{m}}' \curlywedge \bar{\mathsf{m}}& \\
\text{and } \nexists \bar{\mathbf{m}}\boldsymbol{\alpha}? \in \mathtt{op}(\widehat{\bar{\mathbf{m}}}). \bar{\mathbf{m}}\boldsymbol{\alpha}? \approx \bar{\mathsf{m}}'\alpha?& \\
\text{and } \alpha! \neq \surd_{\mathsf{i+1}}& \\
\text{where } \|\bar{\mathsf{m}}|_{O} \cap \widehat{\surd}\| = i\}&
\end{aligned}
$$

Theorem 5 states that a trace-preserving compiler preserves safety properties in the sense of Definition 12.

Theorem 5 (Safety preservation)**.** Let $[\![\cdot]\!]^{\mathcal{S}}_{\mathcal{T}} \in \mathit{TP}$. Let $\mathbf{S}, \widehat{\bar{\mathbf{m}}}$ be such that $\widehat{\bar{\mathbf{m}}} :: \mathbf{S}$. Take $\widehat{\bar{\mathsf{m}}}$ and S such that $\widehat{\bar{\mathsf{m}}} :: \mathsf{S}$ and such that $\widehat{\bar{\mathbf{m}}} \overset{\text{SP}}{\approx} \widehat{\bar{\mathsf{m}}}$. Then, for all $\mathbf{C}$, $\mathbf{TR}(\mathbf{C}) \subseteq \mathbf{S}$ implies $\mathsf{TR}([\![\mathbf{C}]\!]^{\mathcal{S}}_{\mathcal{T}}) \subseteq \mathsf{S}$.

Proof Sketch. Suppose, for the sake of contradiction, that $\mathbf{TR}(\mathbf{C}) \subseteq \mathbf{S}$ but $\mathsf{TR}([\![\mathbf{C}]\!]^{\mathcal{S}}_{\mathcal{T}}) \nsubseteq \mathsf{S}$. Then $\mathsf{TR}([\![\mathbf{C}]\!]^{\mathcal{S}}_{\mathcal{T}})$ must have a trace with a prefix in $\widehat{\bar{\mathsf{m}}}$. We consider two cases. If the prefix contains no invalid input, then by the definition of TP, it must correspond to a source prefix in $\widehat{\bar{\mathbf{m}}}$. Moreover, $\mathbf{TR}(\mathbf{C})$ must have a trace that extends $\widehat{\bar{\mathbf{m}}}$. It follows immediately that $\mathbf{TR}(\mathbf{C}) \nsubseteq \mathbf{S}$. A contradiction. If the prefix has an invalid input, by $[\![\cdot]\!]^{\mathcal{S}}_{\mathcal{T}} \in \mathit{TP}$, the ith such input must be followed by the ith tick from $\widehat{\surd}$ (for every i). Hence, the prefix cannot be in $\widehat{\bar{\mathsf{m}}}$ by $\widehat{\bar{\mathsf{m}}}$'s definition. Again, a contradiction. □

4.2.2 Hypersafety Preservation

Next, we turn to the preservation of hypersafety properties. Unlike safety, which is concerned with single traces, hypersafety is concerned with multiple traces, which lets it capture properties like non-interference. The intuition behind hypersafety is that a set of traces is bad if it has a set of bad prefixes that no good set of traces has. As we can see, the intuition is just like safety, with just one more "level" of sets. Formally, a safety hyperproperty $\mathbb{S}$ (a set of sets of traces) is defined as follows:

$$
\begin{aligned}
&\forall \widehat{\bar{\alpha}} \text{ if } \widehat{\bar{\alpha}} \notin \mathbb{S} \\
&\quad \text{then } (\exists \widehat{\bar{m}}. \widehat{\bar{m}} \leq \widehat{\bar{\alpha}} \text{ and } (\forall \widehat{\bar{\alpha}'}. \text{ if } \widehat{\bar{m}} \leq \widehat{\bar{\alpha}'} \text{ then } \widehat{\bar{\alpha}'} \notin \mathbb{S}))
\end{aligned}
$$

We can characterize every hypersafety property based on the set of set of bad prefixes. We write $\mathbb{M} :: \mathbb{S}$ to mean that $\mathbb{M}$ is the set of all sets of bad prefixes that characterises the safety hyperproperty $\mathbb{S}$.

$$\text{if } \mathbb{M} :: \mathbb{S} \text{ then } \widehat{\bar{\alpha}} \notin \mathbb{S} \text{ iff } \exists \widehat{\bar{m}} \in \mathbb{M}. \widehat{\bar{m}} \leq \widehat{\bar{\alpha}}$$

We define the translation of the set of sets of source bad prefixes by translating all of them under $\approx$. The key technical difference with respect to safety preservation is that we treat as bad singleton sets of all traces in which the ith invalid input is not immediately succeeded by $\surd_{\mathsf{i}}$. The addition of singleton sets is the minimum addition we can make to the set of invalid prefixes to ensure that any (translated) program that contains even one trace wherein a response to an invalid input is not from $\widehat{\surd}$ is considered bad.

This idea of translating hypersafety is formalised in Definition 13.

Definition 13 (Hypersafety relation)**.** Two sets of sets prefixes define the same safety hyperproperty, denoted as $\mathbb{M} \overset{\text{SHP}}{\approx} \mathbb{M}$ if:

$$
\begin{aligned}
\mathbb{M} = \{\widehat{\bar{\mathsf{m}}} \mid \exists \widehat{\bar{\mathbf{m}}} \in \mathbb{M}. \widehat{\bar{\mathsf{m}}} \approx \widehat{\bar{\mathbf{m}}}\} \cup \\
\{\{\bar{\mathsf{m}}\alpha?\alpha!\} \mid \exists \widehat{\bar{\mathbf{m}}} \in \mathbb{M}. \exists \bar{\mathbf{m}} \in \widehat{\bar{\mathbf{m}}}, \bar{\mathsf{m}}'. \bar{\mathbf{m}} \approx \bar{\mathsf{m}}' \curlywedge \bar{\mathsf{m}} \\
\text{and } \nexists \widehat{\bar{\mathbf{m}}'} \in \mathbb{M}. \exists \bar{\mathbf{m}}'\boldsymbol{\alpha}? \in \widehat{\bar{\mathbf{m}}'}. \bar{\mathbf{m}}'\boldsymbol{\alpha}? \approx \bar{\mathsf{m}}'\alpha? \\
\text{and } \alpha! \neq \surd_{\mathsf{i+1}} \\
\text{where } \|\bar{\mathsf{m}}|_{O} \cap \widehat{\surd}\| = i\}
\end{aligned}
$$

Any trace-preserving compiler preserves all hypersafety properties, as Theorem 6 captures.

Theorem 6 (Hypersafety preservation)**.** Let $[\![\cdot]\!]^{\mathcal{S}}_{\mathcal{T}} \in TP$. Let $\mathbb{S}, \mathbb{M}$ be such that $\mathbb{M} :: \mathbb{S}$. Let $\mathbb{M}$ and $\mathbb{S}$ such that $\mathbb{M} :: \mathbb{S}$ and such that $\mathbb{M} \overset{\text{SHP}}{\approx} \mathbb{M}$. Then, for all $\mathbf{C}$, $\mathbf{TR}(\mathbf{C}) \in \mathbb{S}$ implies $\mathsf{TR}([\![\mathbf{C}]\!]^{\mathcal{S}}_{\mathcal{T}}) \in \mathbb{S}$.

The proof follows the same intuition as that of Theorem 5. There is no new fundamental difficulty in proving the theorem.

Remark It is trivial to prove that all safety hyperproperties are preserved under refinement. An intuitive way to understand Theorem 6 is as a generalization of this result to the case where we may have extra actions (invalid inputs) in the target. Basically, Definition 13 strengthens the source property by allowing for some extra behaviour in the target, namely responding to invalid inputs by $\surd$s. *TP* can be seen as a slight weakening of refinement from source to target, that also allows similar extra behaviour in the target. Theorem 6 then says that this weakened form of refinement preserves the strengthened source properties.

4.2.3 Non-Interference Preservation

Theorem 6 states that a *TP* compiler preserves hypersafety properties under a specific translation. An obvious question is whether that translation itself is meaningful, in the sense that it preserves the *intents* of the source hypersafety properties. While a generic answer to this question is impossible to provide (since intent is property-specific), we show here that for a widely considered hypersafety property, namely, non-interference, this is the case under a specific condition on $\approx$. Non-interference is a security policy for information flow control which says that the public (low) outputs of a program must be independent of secret (high) inputs. In other words, in any two traces that agree on all low inputs, all high outputs must also be the same.

To formalize the property, assume that in both the source and the target, inputs and outputs are classified into low and high. Define an equivalence of actions $=_L$ as follows:

$$\begin{array}{c}\textsf{(Low-equiv. on low actions)}\\ \dfrac{\alpha, \alpha' \text{ are low} \qquad \alpha \equiv \alpha'}{\alpha =_L \alpha'}\end{array} \qquad\qquad \begin{array}{c}\textsf{(Low-equiv. on high actions)}\\ \dfrac{\alpha, \alpha' \text{ are high}}{\alpha =_L \alpha'}\end{array}$$

Then, NI can be defined as follows by overloading the $=_L$ notation to lift pointwise to sets of actions.[1]

Definition 14 (NI as a hyperproperty)**.** Recall that $\bar{\alpha}|_I$ and $\bar{\alpha}|_O$ extract inputs and outputs of $\bar{\alpha}$.

$$\begin{aligned}\mathbb{NI} \overset{\mathsf{def}}{=} \{\widehat{\bar{\alpha}} \mid \forall \bar{\alpha}_1, \bar{\alpha}_2 \in \widehat{\bar{\alpha}}.\\ \text{if } \bar{\alpha}_1|_I =_L \bar{\alpha}_2|_I \text{ then } \bar{\alpha}_1|_O =_L \bar{\alpha}_2|_O\}\end{aligned}$$

$\mathbb{NI}$ is a safety hyperproperty. A pair of trace prefixes is bad if the prefixes agree on low inputs but disagree on low outputs. The following theorem shows that a source's $\mathbb{NI}$ when translated as described in Theorem 6 yields a hyperproperty that is contained in the target's $\mathbb{NI}$, if $\approx_O$ satisfies a specific *injectivity* condition. This immediately implies that if a source program satisfies $\mathbb{NI}$ then compiling it through a *TP* compiler yields a program that satisfies $\mathbb{NI}$. The injectivity condition means that every source symbol is related to a unique target symbol. This is required to prevent the compiler from encoding secrets in different representations of the same low output. Additionally, all elements of $\widehat{\surd}$ are considered to be observable, i.e., tagged as low.

Definition 15 (Injectivity)**.** We say that $\approx$ is injective if $\boldsymbol{\alpha} \approx \alpha_1$ and $\boldsymbol{\alpha} \approx \alpha_2$ imply $\alpha_1 = \alpha_2$.

Theorem 7 (Non-interference is preserved)**.** Let $\mathbb{M} :: \mathbb{NI}$ and $\approx_O$ be injective. Let $\mathbb{M} \overset{\mathrm{SHP}}{\approx} \mathbb{M}$ and let $\mathbb{S}$ be a hyperproperty such that $\mathbb{M} :: \mathbb{S}$. Then, $\forall \widehat{\bar{\alpha}} \in \mathbb{S}, \widehat{\bar{\alpha}} \in \mathbb{NI}$.

4.3 Limitations of TPC and Secure Compilation

This section discusses how **TPC** can preserve liveness and why it cannot preserve hyperliveness and arbitrary hyperproperties in general.

[1] This is just one possible definition of NI in a reactive setting. See the work of Bohannon *et al.* [16] for a detailed discussion of definitions of NI in a reactive setting.

4.3.1 Preserving Liveness

Liveness properties specify (good) events that should eventually occur. Formally, a liveness property L (a set of traces) is defined as follows.

$$\forall \bar{m}.\ \exists \bar{\alpha}.\ \bar{m} \leq \bar{\alpha} \text{ and } \bar{\alpha} \in L$$

In general, existing secure compilers do not aim to provide any liveness properties since an adversary can always prevent the compiled program from achieving its intended goal by continuously providing it bad inputs. And of course, assuming that the attacker will not provide invalid inputs would make the work security-irrelevant.

Trace-preserving compilation as presented does not aim to preserve liveness properties. For example, consider a trivial source program that always produces a $\mathbf{0}$ in response to any input. Consider the source liveness property $\mathbf{L}$:"produce an infinite number of $\mathbf{0}$s". The source program satisfies $\mathbf{L}$. Suppose the program is compiled to a target language where $\mathbf{0}$ is mapped to $\mathsf{0}$, and where there is at least one invalid input. The source liveness property $\mathbf{L}$ would intuitively translate to the target liveness property L:"produce an infinite number of $\mathsf{0}$s". However, no trace-preserving compiler can enforce this target property since the environment can always starve the program by continuously providing invalid inputs to which the compiled program must respond by producing elements of $\widehat{\surd}$ and never a $\mathsf{0}$.

Trace-preserving compilation, in its disregarding variant, can attain a form of liveness if we assume that the attacker is fair, i.e., that eventually he will provide a valid input. As for safety, we would need to change the notion of liveness, to allow the interleaving of invalid actions followed by a $\surd$. However, this would alter the meaning of the target level liveness property much more than what happens with safety. Let us consider the liveness property $\mathbf{L}$ above. Under this translation, it would be translated into L': "produce an infinite number of $\mathsf{0}$ *interleaved with any number of* $\surd$". The insertion of $\surd$s is necessary, as after any $\mathsf{0}$ output the attacker can provide invalid inputs, to which the compiled component must respond with a $\surd$. This would change the meaning slightly, but in certain cases it would be acceptable. Concerning preservation of liveness, the fairness assumption would ensure that eventually the attacker will supply a valid action. By definition of **TPC**, the compiled code will respond respond to it with a $\mathsf{0}$, as meant by the source property $\mathbf{L}$.

This idea of translating liveness is formalised in Definition 17, though it relies on fairness of attackers which is defined below.

Definition 16 (Fair attacker)**.** A trace $\bar{\alpha}$ is fair, denoted as $\mathsf{fair}_{\approx}(\bar{\alpha})$, if any $\surd$ in it is eventually followed by a valid input according to relation $\approx$.

$$\mathsf{fair}_{\approx}(\bar{\alpha}) \overset{\mathsf{def}}{=} \text{if } \bar{\alpha} \equiv \bar{\mathsf{m}}\alpha?\surd\bar{\alpha}' \text{ then } \exists\alpha \in \bar{\alpha}'|_{I}.\exists\boldsymbol{\alpha} \approx \alpha$$

Definition 17 (Liveness relation)**.** Two sets traces define the same liveness property, denoted as $\widehat{\bar{\mathbf{L}}} \overset{\mathrm{LP}}{\approx} \widehat{\bar{\mathsf{L}}}$ if:

$$\widehat{\bar{\mathsf{L}}} \overset{\mathsf{def}}{=} \{\bar{\alpha} \mid \mathsf{fair}_{\approx}(\bar{\alpha}) \text{ and } \forall \bar{\alpha}' \curlywedge \bar{\alpha}.\ \exists \bar{\boldsymbol{\alpha}} \in \widehat{\bar{\mathbf{L}}}.\ \bar{\boldsymbol{\alpha}} \approx \bar{\alpha}'\}$$

A trace-preserving compiler that implements the disregarding variant preserves liveness properties, as Theorem 8 captures.

Theorem 8 (Liveness preservation)**.** Let $[\![\cdot]\!]^{\mathcal{S}}_{\mathcal{T}} \in TP$. Let $\widehat{\bar{\mathbf{L}}}, \widehat{\bar{\mathsf{L}}}$ be such that $\widehat{\bar{\mathbf{L}}} \overset{\mathrm{LP}}{\approx} \widehat{\bar{\mathsf{L}}}$. Then, for all $\mathbf{C}$, $\mathbf{TR}(\mathbf{C}) \subseteq \widehat{\bar{\mathbf{L}}}$ implies $\mathsf{TR}([\![\mathbf{C}]\!]^{\mathcal{S}}_{\mathcal{T}}) \subseteq \widehat{\bar{\mathsf{L}}}$.

4.3.2 Preserving Hyperliveness and Arbitrary Hyperproperties

While the fairness assumption seems sufficient to achieve (a form of) liveness preservation for *TP*, it does not seem to suffice for general hyperliveness preservation. Moreover, there does not seem to be a property that would help preserving hyperliveness. In fact, hyperliveness includes properties such as "the average response time is below 1 second" or "the average response time is above 4 seconds". Finding a uniform, general way to enforce all these properties without relying on their specific statements does not seem feasible.

As hyperliveness are a subclass of all hyperproperties, secure compilers (and trace-preserving compilers) cannot preserve arbitrary hyperproperties. Investigating the preservation of specific hyperproperties, e.g., by letting the compiler take the hyperproperty as input, seems feasible and it is left for future work.

5 Trace-Preserving and Fully Abstract Compilation

This section proves that trace preservation and full abstraction are not equivalent: a *TP* compiler is *FA* but not vice-versa (Section 5.1). Then, it defines same-fail behaviour (*FSB*), an additional property that, if satisfied by a correct compiler, implies that the compiler also has TP^{H}, the halting variant of **TPC** (Section 5.2).

5.1 Relation between *TP* and *FA*

In order to relate the two notions, for the rest of this section assume that $\approx$ is injective as in Definition 15. This is not an unrealistic assumption, since all existing fully abstract compilers based on contextual equivalence satisfy and, in fact, it seems that writing a fully abstract meaningful compiler without this assumption may be impossible, as illustrated in the following example.

Example 6 (*FA* requires injectivity)**.** Consider the source language of Example 3 and exactly two programs $\lambda\mathbf{x.true}$ and $\lambda\mathbf{x.(true} \vee \mathbf{false)}$ that implement the constant function that returns **true**. These two programs are (trivially) trace-equivalent. Suppose this language is compiled to $\lambda^{\mathbb{N}}$ and that, for the sake of argument, the relation $\approx$ is not injective: it relates $\mathbf{false} \approx 0$ and $\mathbf{true} \approx 1, 2, \ldots$. Consider a compiler that maps the two source functions to $\lambda x.1$ and $\lambda x.2$, respectively. By having multiple mappings for $\mathbf{true}$ the target equivalence is broken and this compiler is trivially not *FA*: the two target programs are not trace equivalent, even though the two source programs are. ⊡

With injectivity, trace-preserving compilation implies fully abstract compilation (Theorem 9).

Theorem 9 (*TP* implies *FA*)**.** $\forall [\![\cdot]\!]^{\mathcal{S}}_{\mathcal{T}}, [\![\cdot]\!]^{\mathcal{S}}_{\mathcal{T}} \in \mathit{TP} \Rightarrow [\![\cdot]\!]^{\mathcal{S}}_{\mathcal{T}} \in \mathit{FA}$.

The converse of Theorem 9 is false because there are fully abstract compilers that are not trace-preserving.

Theorem 10 (*FA* does not imply *TP*)**.** $\exists [\![\cdot]\!]^{\mathcal{S}}_{\mathcal{T}} \in \mathit{FA}.\ [\![\cdot]\!]^{\mathcal{S}}_{\mathcal{T}} \notin \mathit{TP}$.

Proof. There are many compilers that are in *FA* but not in *TP*. The compiler of Example 2 is one such example. Here, we present a second, non trivial example. Consider a source language λ^{B}, which is a generalisation of the language of Example 1 that allows arbitrary operations on booleans and the target language $\lambda^{\mathbb{N}}$ from Example 3, which now includes min and max operations on natural numbers. While both languages are λ-calculi with higher-order functions, we restrict top-level programs in λ^{B} to input and output booleans, i.e., to the type $\mathtt{Bool} \to \mathtt{Bool}$. Similarly, top-level $\lambda^{\mathbb{N}}$ programs input and output numbers, i.e., they have the type $\mathbb{N} \to \mathbb{N}$. The top-level programs of the two languages are denoted $\mathbf{C}$ and C, respectively. The type system of both languages is omitted for brevity, but the syntax is shown below.

$$\mathbf{C} ::= \mathbf{f}(\mathbf{x}) = \mathbf{t} \qquad\qquad \mathsf{C} ::= \mathsf{f}(\mathsf{x}) = \mathsf{e}$$

$$\mathbf{t} ::= \mathbf{true} \mid \mathbf{false} \mid \mathbf{x} \mid \mathbf{t}\ \mathbf{t} \mid \lambda\mathbf{x} : \tau.\mathbf{t} \mid \mathbf{t} \wedge \mathbf{t} \mid \mathbf{t} \vee \mathbf{t} \mid \mathbf{f}$$

$$\mathsf{e} ::= \mathsf{n} \in \mathbb{N} \mid \mathsf{x} \mid \mathsf{e}\ \mathsf{e} \mid \lambda\mathsf{x}.\mathsf{e} \mid \mathtt{min}(\mathsf{e}, \mathsf{e}) \mid \mathtt{max}(\mathsf{e}, \mathsf{e}) \mid \mathsf{f}$$

Both languages follow a call-by-value reduction which is straightforward but for the evaluation of $\mathtt{min}()$ and $\mathtt{max}()$. The former follows Rule $\lambda^{\mathbb{N}}$-eval-min as presented below, while the latter is analogous.

$$\frac{\begin{array}{c}\text{if } \mathsf{v}_1 \in \mathbb{N} \text{ then } v_1 = \mathsf{v}_1 \text{ else } v_1 = 1\\ \text{if } \mathsf{v}_2 \in \mathbb{N} \text{ then } v_2 = \mathsf{v}_2 \text{ else } v_2 = 1\\ \text{if } v_1 > v_2 \text{ then } \mathsf{v} = v_1 \text{ else } \mathsf{v} = v_2\end{array}}{\mathtt{min}(\mathsf{v}_1, \mathsf{v}_2) \hookrightarrow \mathsf{v}}\ (\lambda^{\mathbb{N}}\text{-eval-min})$$

The relation between the languages is that of Example 3: it includes $\mathbf{true} \approx 1$ and $\mathbf{false} \approx 0$ and is defined inductively on other terms based on their type.

Consider the two-step compiler $[\![\cdot]\!]^{\lambda^{B}}_{\lambda^{\mathbb{N}}}$ from λ^{B} to $\lambda^{\mathbb{N}}$ shown in Figure 2. The compiler maps $\mathtt{Bool}$ to $\mathbb{N}$. At the top-level, in the translation of $\mathbf{f}(\mathbf{x}) = \mathbf{t}$, it modifies the input x to $\mathtt{min}(\mathsf{x}, 1)$ before passing it to the translation of $\mathbf{t}$. So, the translation of $\mathbf{t}$ only receives valid inputs (0 or 1).

$$[\![\mathbf{f}(\mathbf{x}) = \mathbf{t}]\!]^{\lambda^B}_{\lambda^{\mathbb{N}}} = (\mathsf{f}(\mathsf{x}) = [\![\mathbf{t}]\!]^{\lambda^B}_{\lambda^{\mathbb{N}}}[\mathtt{min}(\mathsf{x}, 1)/\mathsf{x}])$$

$$\begin{aligned}
[\![\mathbf{true}]\!]^{\lambda^B}_{\lambda^{\mathbb{N}}} &= 1 & [\![\mathbf{x}]\!]^{\lambda^B}_{\lambda^{\mathbb{N}}} &= \mathsf{x} \\
[\![\mathbf{false}]\!]^{\lambda^B}_{\lambda^{\mathbb{N}}} &= 0 & [\![\mathbf{f}]\!]^{\lambda^B}_{\lambda^{\mathbb{N}}} &= \mathsf{f} \\
[\![\lambda\mathbf{x} : \tau.\mathbf{t}]\!]^{\lambda^B}_{\lambda^{\mathbb{N}}} &= \lambda\mathsf{x}.[\![\mathbf{t}]\!]^{\lambda^B}_{\lambda^{\mathbb{N}}} & [\![\mathbf{t}\ \mathbf{t}']\!]^{\lambda^B}_{\lambda^{\mathbb{N}}} &= [\![\mathbf{t}]\!]^{\lambda^B}_{\lambda^{\mathbb{N}}}\ [\![\mathbf{t}']\!]^{\lambda^B}_{\lambda^{\mathbb{N}}}
\end{aligned}$$

$$\begin{aligned}
[\![\mathbf{t_1} \wedge \mathbf{t_2}]\!]^{\lambda^B}_{\lambda^{\mathbb{N}}} &= \mathtt{min}([\![\mathbf{t_1}]\!]^{\lambda^B}_{\lambda^{\mathbb{N}}}, [\![\mathbf{t_2}]\!]^{\lambda^B}_{\lambda^{\mathbb{N}}}) \\
[\![\mathbf{t_1} \vee \mathbf{t_2}]\!]^{\lambda^B}_{\lambda^{\mathbb{N}}} &= \mathtt{max}([\![\mathbf{t_1}]\!]^{\lambda^B}_{\lambda^{\mathbb{N}}}, [\![\mathbf{t_2}]\!]^{\lambda^B}_{\lambda^{\mathbb{N}}})
\end{aligned}$$

Figure 2: Example of a fully abstract compiler.

It is straightforward to prove that $[\![\cdot]\!]^{\lambda^B}_{\lambda^{\mathbb{N}}}$ is correct and that it is fully abstract (as proven in the appendix). However, the compiler is not trace-preserving. On an invalid input like 2 a compiled program still produces either 0 or 1, both of which correspond to source values and, hence, are not $\surd$s.

In fact, this compiler does not preserve all safety properties in the sense of Theorem 5. Consider the source program $\mathbf{f}(\mathbf{x}) = \mathbf{x}$. This program satisfies the safety property "Output **true** only in response to **true**." Its translation $\mathsf{f}(\mathsf{x}) = \mathtt{min}(\mathsf{x}, 1)$ does not satisfy the translation of this safety property. In fact, it outputs 1 (the translation of **true**) in response to input 4. The translation of the safety property implied in Theorem 5 will require that the program produce $\surd$ in response to the input 4. □

5.2 *FA* Can Imply TPC

Unlike the fully abstract compiler of Example 2, existing fully abstract compilers respond to invalid inputs in a way that is completely distinct from outputs produced in response to valid inputs. These compilers either prevent invalid inputs by using a target-level type system [8, 7, 17, 25] or with runtime checks [40, 5, 28, 21] that halt the program completely.

We use the term "fail-safe behaviour" (*FSB*) to mean that the program halts (stutters with non-source outputs forever) after an invalid input.

Definition 18 (Fail-safe-behaviour compiler)**.** $[\![\cdot]\!]^{\mathcal{S}}_{\mathcal{T}} \in FSB \overset{\mathsf{def}}{=} \forall\mathbf{C}.\ \forall\bar{\alpha} \in \mathsf{TR}([\![\mathbf{C}]\!]^{\mathcal{S}}_{\mathcal{T}})$. if $\nexists\bar{\boldsymbol{\alpha}} \in \mathbf{TR}(\mathbf{C}).\bar{\boldsymbol{\alpha}} \approx \bar{\alpha}$, then $\bar{\alpha} \equiv \bar{\mathsf{m}}_1\alpha?\surd\bar{\alpha}_2$ and $\exists\bar{\mathbf{m}}_{\mathbf{1}} \in \mathtt{obs}(\mathbf{TR}(\mathbf{C})).\ \bar{\mathbf{m}}_{\mathbf{1}} \approx \bar{\mathsf{m}}_1$ and $\nexists\boldsymbol{\alpha}? \approx \alpha?$ and $\nexists\surd \in \bar{\mathsf{m}}_1$ and $\bar{\alpha}_2|_O = \surd$.

FSB is very similar to the halting version of **TPC**. Other definitions of invalid traces can be used as well. We formalise *FSB* this way since it is the one most similar to what existing secure compilers do. We rely on this definition to prove that a compiler that is both *FSB* and *CC* is TP^H (Theorem 11).

As existing fully abstract compilers are generally also correct, we claim that what existing fully abstract compilers really achieve is TP^H. Specifically, compiler full abstraction forces anyway the code to perform checks on all inputs.

The *FSB* condition really only ensures that the checks deal with invalid input in a uniform way to prevent the problems of Example 3.

Theorem 11 (Correctness and fail-safe behaviour imply trace-preservation)**.** $\forall[\![\cdot]\!]^{\mathcal{S}}_{\mathcal{T}}$. if $[\![\cdot]\!]^{\mathcal{S}}_{\mathcal{T}} \in CC$ and $[\![\cdot]\!]^{\mathcal{S}}_{\mathcal{T}} \in FSB$ then $[\![\cdot]\!]^{\mathcal{S}}_{\mathcal{T}} \in TP^{H}$.

Again, readers familiar with existing literature on fully abstract compilation may argue that this property is hardly applicable to existing work on fully abstract compilation. Here, we have defined this property in terms of traces simply to be able to describe its implications in our formal setting. Section 6.3 describes a non-trace-based definition of *FSB* that is amenable to existing work on fully abstract compilation.

6 Beyond Reactive Programs

This section sets up the formal background (Section 6.1) and discusses how to scale the presented results to non-reactive settings, i.e., to settings that are more commonly found in existing secure compilation work (Section 6.2). Then it rephrases *FSB* without using traces and proves that an existing fully abstract compiler also has *FSB*, so it attains the halting variant of **TPC** (Section 6.3).

6.1 Formal Tools for Non-Reactive Languages

Many existing work on secure compilation is in the sequential programs setting [7, 8, 17, 5, 28, 30, 25, 40, 42, 32]. In these cases, program are related via a notion of contextual equivalence (Section 6.1.1) or well-behaved contextual equivalence (Section 6.1.2). The behaviour of programs can also be described via traces (Section 6.1.3), but additional properties need to be proven in order for this reasoning to be meaningful.

6.1.1 Contextual Equivalence

Contextual equivalence is the coarsest program equivalence that the operational semantics of a language yield [44]. It is used to reason about programs of the same language.

As the name suggests, contextual equivalence relies upon the notion of *context*. A (program) context is a partial program with a hole ($[\cdot]$), so it follows the same syntax and typing (if any) of programs. Formally, $\mathbb{C} \overset{\textsf{def}}{=} C[\cdot]$. The hole in the context can be filled by another program; this results in a whole program that can be executed according to the language semantics.

Informally, two *partial programs* C are contextually equivalent if they *have the same behaviour* for any possible context $\mathbb{C}$ that they are plugged to. Having *"the same behaviour"* means: *"according to the operational semantics of the language, the two programs cannot be distinguished just by looking at the results"*.

Definition 19 (Contextual equivalence [44]). $C_1 \simeq_{ctx} C_2 \overset{\mathsf{def}}{=} \forall \mathbb{C}, \mathbb{C}[C_1]\Uparrow \iff \mathbb{C}[C_2]\Uparrow$, where $\Uparrow$ indicates divergence, i.e., the execution of an unbounded number of reduction steps. Divergence can be replaced by termination (i.e., $\Downarrow$, as necessary for strictly-terminating calculi); the two formulations are equivalent.

6.1.2 Well-Behaved Contexts

Reasoning about programs written in different languages is often done by means of a cross-language relation $\sim$. Unlike relations $\approx_I$ and $\approx_O$ (from Section 2.2), relation $\sim$ is not only defined on inputs and outputs, but on all language-related elements (values, terms, contexts etc). Often such a relation is instantiated with a cross-language logical relation [27, 13, 14, 6, 35, 37].

$\sim$ lets us define the set of well-behaved target contexts w.r.t. the source language, i.e., target-level contexts that have a source-level counterpart (Definition 20).

Definition 20 (Well-behaved $\mathcal{T}$ contexts w.r.t. $\mathcal{S}$). $WBctx^{\mathcal{S}}_{\mathcal{T}} = \{\mathsf{C} \mid \exists \mathbf{C}.\, \mathbf{C} \sim \mathsf{C}\}$

Example 7 (Well-behaved contexts). Consider the source language of Example 2 and the target language to be $\lambda^{\mathbb{N}}$. Assume $\sim$ contains the following: $\mathbf{true} \sim 1$ and $\mathbf{false} \sim 0$.

Consider context $[\cdot]\ 1$, whose hole expects a function; this is well-behaved, since it is related to the source-level context $[\cdot]\ \mathbf{true}$. On the other hand, context $[\cdot]\ 3$ is not well behaved. No source-level context relates to it, as no source-level context ever applies a value related to 3 to a function. ⊡

6.1.3 Trace Semantics

Trace equivalence is another tool to reason about partial programs written in the same language, and it is simpler than contextual equivalence [41, 40, 5, 28, 29]. Trace equivalence relates two components that exhibit the same trace semantics, i.e., whose behaviour can be described with the same set of traces (as in Definition 2).

Formally, in the sequential setting, a trace semantics is a triple: $\mathsf{TR} \overset{\mathsf{def}}{=} \{\Sigma; \alpha; \xrightarrow{\bar{\alpha}}\!\!\!\!\rightarrow\}$. Σ is the set of states of the trace semantics. Σ must include two kinds of states: operational semantics states and "unknown" ones. The former models that the execution is within the component while the latter models that the execution is outside of it. α are the actions that can be generated by the semantics, they follow the same formalisation of actions and traces presented in Definition 2. $\xrightarrow{\bar{\alpha}}\!\!\!\!\rightarrow \subseteq \Sigma \times \bar{\alpha} \times \Sigma$ is a relation that specifies how actions are concatenated into traces $\bar{\alpha}$.

The trace semantics of a program C, indicated as $TR(C)$, is the set of traces it can generate from its starting state $\Sigma^0(C)$.

Definition 21 (Trace semantics). $TR(C) \overset{\mathsf{def}}{=} \{\bar{\alpha} \mid \exists \Sigma. \Sigma^0(C) \xrightarrow{\bar{\alpha}}\!\!\!\!\rightarrow \Sigma\}$.

Two programs are trace equivalent if their trace semantics coincide, as already formalised in Definition 3.

When a language is defined, its operational semantics yields contextual equivalence. When a language is also given a trace semantics, the resulting notion of trace equivalence must be shown to coincide with contextual equivalence, otherwise any reasoning based on traces can be meaningless. Formally, this is called full abstraction of the trace semantics (Definition 22).[2] Let *FAT* be the set of all fully abstract trace semantics.

Definition 22 (Fully abstract trace semantics). $\mathsf{TR} \in \mathit{FAT} \overset{\mathsf{def}}{=} \forall C_1, C_2.\ C_1 \simeq_{ctx} C_2 \iff C_1 \overset{\mathrm{T}}{=} C_2$

The simplest way to develop a fully abstract trace semantics is by construction, i.e., it can be devised semi-mechanically from the operational semantics. However this is not always possible nor simple, so devising a fully abstract trace semantics for complex systems is an active research topic [29, 41, 31, 47].

6.2 Non-Reactive Trace-Preserving Compilation

To make **TPC** meaningful in a non-reactive setting, both the source and the target languages must have fully abstract trace semantics, with which their hyperproperties are expressed. This can be expressed through two assumptions.

Assumption 1 (The source-level trace semantics TR is fully abstract). $\mathsf{TR} \in \mathit{FAT}$.

Assumption 2 (The target-level trace semantics TR is fully abstract). $\mathsf{TR} \in \mathit{FAT}$.

Assumption 2 does not need to hold in general but just for compiled components, i.e., for a subset of the programs of $\mathcal{T}$, the target language.

No existing work on secure compilation satisfies both these assumptions as no existing work was interested in understanding the connection to hyperproperties. Some existing work, however, satisfies Assumption 2, as they equip the target language with fully abstract trace semantics for simplifying the proof of full abstraction [28, 40, 42, 41, 32].

6.2.1 Non-Reactive Fail-Safe Behaviour

Definition 23 redefines *FSB* from Definition 18 without traces. Let $\mathbb{C} \frown \mathsf{C}$ indicate that $\mathbb{C}$ and C are compatible, so C can fill the hole of $\mathbb{C}$.

Definition 23 (Fail-safe-behaviour compiler (without traces)). $[\![\cdot]\!]^{\mathcal{S}}_{\mathcal{T}} \in \mathit{FSB} \overset{\mathsf{def}}{=} \forall \mathbb{C} \notin \mathit{WBctx}^{\mathcal{S}}_{\mathcal{T}}, \forall \mathbf{C}.$ if $\mathbb{C} \frown [\![\mathbf{C}]\!]^{\mathcal{S}}_{\mathcal{T}}$, then $\mathbb{C}[[\![\mathbf{C}]\!]^{\mathcal{S}}_{\mathcal{T}}] \hookrightarrow^{*} \mathsf{t}$ and $\exists \mathsf{t}_{\mathsf{stuck}}.\ \mathsf{t} \simeq_{ctx} \mathsf{t}_{\mathsf{stuck}}$ and $\forall \mathbb{C}'.\ \mathbb{C}'[\mathsf{t}_{\mathsf{stuck}}] \not\hookrightarrow$.

[2] Standard terminology may be confusing since full abstraction is used for both compilers and semantics. When the qualifier 'compiler' or 'semantics' is omitted, it should be clear from the context which notion is meant.

The intuition for Definition 23 is that for any invalid interaction (i.e., an interaction with a context $\mathbb{C}$ that is not well-behaved) a compiled component reduces to a term t that is equivalent to a term $\mathsf{t}_{\mathsf{stuck}}$ that cannot reduce any further. $\mathsf{t}_{\mathsf{stuck}}$, and its equivalent terms, are what correspond to $\surd$. This models what existing fully abstract compilers do, i.e., they halt when some invalid interaction is detected. Instead of enforcing that t is stuck, we require it to be equivalent to a stuck term to model, for example, reduction to a function that will get stuck when it is applied.

6.3 TPC for an Existing Fully Abstract Compiler

We believe that **TPC** is actually what all existing fully abstract compilers achieve. This section argues in favour of this belief by proving that at least one fully abstract compiler of Devriese *et al.* [21] is *FSB* according to Definition 23. Thus, if fully abstract trace semantics were given to the languages (to reason about hyperproperties), it would be *TP*. We take this compiler since it is simple enough to prove Definition 23 easily.

We believe that this property holds for most existing fully abstract compilers because those works can be split in two main sets. The first ones are those that rely on typed target languages [7, 8, 17, 38], which trivially satisfy *FSB* because the typed target contexts that would violate compiled components cannot be composed with it (so they do not satisfy $\mathbb{C} \frown [\![\mathbf{C}]\!]^{\mathcal{S}}_{\mathcal{T}}$). The second ones are those that rely on untyped target languages [25, 40, 5, 28, 30], which mostly follow the same general principle of the compiler we examine: they perform some form of runtime checking and behave uniformly when they fail by terminating. All of the existing fully abstract compilers are also correct.

For the compiler of this section, the source language λ^{τ} is a simply-typed λ-calculus with booleans and unit, and a fix operator while the target language λ^{u} is an untyped λ-calculus with booleans and unit. The operational semantics of both languages is mostly unsurprising and, as the syntax, in large part omitted. The only interesting cases are the reduction of fix in λ^{τ} and how λ^{u} treats non well-formed arguments, which are presented below (the latter is presented only in the case of sequencing and if-then-else).

$$\frac{}{\begin{array}{l}\mathrm{fix}_{\tau_1 \to \tau_2}\ (\lambda \mathbf{x} : \tau_1 \to \tau_2 .\, \mathbf{t}) \hookrightarrow \\ \quad \mathbf{t}[(\lambda\ \mathbf{y} : \tau_1 .\, \mathrm{fix}_{\tau_1 \to \tau_2}\ (\lambda \mathbf{x} : \tau_1 \to \tau_2 .\, \mathbf{t})\ \mathbf{y})/\mathbf{x}]\end{array}}\ (\lambda^{\tau}\text{-Eval-fix})$$

$$\frac{\begin{array}{c}\mathsf{v} \equiv \mathsf{unit} \Rightarrow \mathsf{t}' \equiv \mathsf{t} \\ \mathsf{v} \not\equiv \mathsf{unit} \Rightarrow \mathsf{t}' \equiv \mathsf{wrong}\end{array}}{\mathsf{v};\mathsf{t} \hookrightarrow \mathsf{t}'}\ (\lambda^{\mathsf{u}}\text{-Eval-seq-next}) \qquad \frac{\begin{array}{c}\mathsf{v} \equiv \mathsf{true} \Rightarrow \mathsf{t}' \equiv \mathsf{t}_1 \\ \mathsf{v} \equiv \mathsf{false} \Rightarrow \mathsf{t}' \equiv \mathsf{t}_2 \\ (\mathsf{v} \not\equiv \mathsf{true} \wedge \mathsf{v} \not\equiv \mathsf{false}) \\ \Rightarrow \mathsf{t}' \equiv \mathsf{wrong}\end{array}}{\text{if } \mathsf{v} \text{ then } \mathsf{t}_1 \text{ else } \mathsf{t}_2 \hookrightarrow \mathsf{t}'}\ (\lambda^{\mathsf{u}}\text{-Eval-if-v})$$

$[\![\cdot]\!]^{\lambda^{\tau}}_{\lambda^{\mathsf{u}}}$ is the fully abstract compiler from λ^{τ} and λ^{u}; it is defined as follows:

$$\text{if } \mathbf{t} : \tau \text{ then } \quad [\![\mathbf{t}]\!]^{\lambda^{\tau}}_{\lambda^{\mathsf{u}}} = \mathsf{protect}_{\tau}\mathtt{erase}(\mathbf{t})$$

where `erase`() is a type-erasing function and $\mathsf{protect}$ is a dynamic typechecker on arguments received from the context whose definition is moved to the appendix for space reasons. Any argument that does not respect the expected structure will cause $\mathsf{protect}$ to reduce to wrong, without executing the securely-compiled code. Intuitively, wrong is the $\surd$ (or the term t from Definition 23).

The compiler is correct (Theorem 12) and *FSB* (Theorem 13), so it is trace-preserving (Theorem 14).

Theorem 12 ($\llbracket \cdot \rrbracket^{\lambda^{\tau}}_{\lambda^{u}}$ is correct)**.** $\llbracket \cdot \rrbracket^{\lambda^{\tau}}_{\lambda^{u}} \in CC$.

Proof. See [21]. □

Theorem 13 ($\llbracket \cdot \rrbracket^{\lambda^{\tau}}_{\lambda^{u}}$ has fail-safe behaviour)**.** $\llbracket \cdot \rrbracket^{\lambda^{\tau}}_{\lambda^{u}} \in FSB$.

Proof Sketch. Intuitively, all ill-behaved interactions either reduce to wrong immediately or reduce to functions whose body will reduce to wrong once an argument is supplied. Consider the term $\mathbf{true}$; its compilation is $(\lambda \mathsf{x}.\, \mathsf{x})\ \mathsf{true}$. Consider the following non-well-behaving context for the term above which tries to use it as a function instead of as a Boolean: $\mathbb{C} = [\cdot]\ \mathsf{true}$. Once the compiled term is plugged into $\mathbb{C}$, the resulting term performs the following reductions:

$$((\lambda \mathsf{x}.\, \mathsf{x})\ \mathsf{true})\ \mathsf{true} \hookrightarrow \mathsf{true}\ \mathsf{true} \hookrightarrow \mathsf{wrong}$$

□

Theorem 14 ($\llbracket \cdot \rrbracket^{\lambda^{\tau}}_{\lambda^{u}}$ is trace-preserving)**.** $\llbracket \cdot \rrbracket^{\lambda^{\tau}}_{\lambda^{u}} \in TP^{H}$.

7 Related Work

Secure compilation has been mostly formalized in the forms of compiler full abstraction and non-interference preservation.

Compiler full abstraction was introduced by Abadi [1], to argue against the compilation of inner classes in an early version of the JVM and for a way to compile the π calculus into the SPI-calculus. Papers that prove full abstraction achieve this by relying on different target-language features: type systems [8, 7, 17], cryptographic primitives [18, 20, 2], memory protection techniques [5, 28, 40, 42] and dynamic checks [21, 25]. Two main approaches exist to proving compiler full abstraction: cross-language logical relations [8, 17, 7, 21] and target-level trace semantics [28, 40, 29]. Concerning the properties of full abstraction, the conditions that make fully abstract compilation between two languages possible have been identified by Parrow [39]. Gorla and Nestmann [26] concluded that full abstraction is meaningful only when it entails properties such as security, thus supporting the motivation for our work. No existing work relates full abstraction (or secure compilation) with hyperproperties in general. The closest pieces of related to hyperproperty preservation are those whose proof is based on trace semantics, as they already fulfil some of the requirements of **TPC**.

Some prior work [10, 11, 12] provides secure compilers that preserve specific hyperproperties, notably non-interference. In all cases, the target language is assumed to be well-typed. Since both the source and target type systems imply non-interference, compiler type preservation implies non-interference preservation. Tse and Zdancewic present a non-interference-preserving secure compiler from the dependency core calculus (DCC) to System F [46].

Recently, secure compartmentalizing compilation (SCC) has also been proposed as a criterion for secure compilation, as it addresses some limitations of "vanilla" compiler full abstraction related to modularity [30]. The main differences between **TPC** and SCC are that (i) SCC considers non-deterministic source languages, and (ii) SCC enforces a fixed structure on the components that encompasses the whole source program. SCC also considers modular compilers, but **TPC** can scale to modular compilers (see Appendix C). In the SCC work, the authors suggest a way to turn compiler full abstraction into SCC by addressing the aforementioned issues (plus one about modularity). We believe that the same approach can be taken with **TPC** to ensure that it also implies SCC. Concerning (i), the non-determinism in source languages for SCC compilers is restricted to affect just the component where the non-determinism happens. Concerning (ii), the notion of plugging programs and contexts is made to adhere to a given shape (or interface), which specifies how the rest of the program is compartimentalised. We believe that by adding the same restriction to the source languages, **TPC** can scale to non-deterministic languages and to programs that have a stipulated structure.

TPC (and more generally secure compilation) also bears a close connection with security policies enforcement by means of runtime monitors The literature on enforcing security properties has a number of suggestion for automata that enforce safety properties. The seminal work of Schneider [45] defined truncation automata, which terminate a program when an undesired action is encountered. Then, Ligatti *et al.* [34] defined suppression automata, which prevent a certain program behaviour but does not alter program behaviour otherwise. The latter kind of automata were further studied by Bielova and Massacci [15] in the context of suppressing behaviour but resuming from a good program state. The halting variant of **TPC** can be seen as a truncating automaton wrapped around compiled code while the disregarding one can be seen as a suppression automaton wrapped around compiled code. However, none of these work discusses automata in a cross-language setting, nor did they need a special action such as $\surd$ to relate cross-language (hyper)properties and their meaning as **TPC** does. Additionally, Ligatti *et al.* [34] also defined insertion automata, which silently replace invalid behaviour with valid ones with a sort of sanitisation pass. Albeit the definition of **TPC** does not directly consider input sanitisation, if the relation $\approx$ were adapted to relate source inputs to target inputs *after* sanitization, then it could be seen as an insertion automata. We leave the formal details to future work.

Trace semantics have been used to reason about the security properties of many reactive systems [49, 24]. We believe that such work can be a good starting point for understanding how to expand the results of this paper to nondeterministic systems.

In this work, as in many others on fully abstract compilation, we consider possible optimisations to be a part of $[\![\cdot]\!]^{\mathcal{S}}_{\mathcal{T}}$. Thus, these results can be made to scale to optimizing compilers as long as the optimisations respect the assumptions needed by **TPC**. The work of D'Silva *et al.* [23] studies the problem of securing compiler optimisation as a separate phase, clearly identifying which compiler optimisations would violate such assumptions.

8 Conclusion

This paper presented a correctness criterion for secure compilation. We show that this criterion preserves safety and hypersafety properties under suitable source-to-target translations of the the properties. We show that the criterion, **TPC**, is stronger than full abstraction for compilers, but can be attained with little more effort beyond that needed to attain full abstraction. At least one existing fully abstract compiler already attains **TPC**.

We believe that this paper clarifies what secure compilation means in terms of preservation of security-relevant (hyper)properties. Additionally, it clarifies the limitations and relevance of fully abstract compilation in the context of security.

Acknowledgements: we would like to thank Amal Ahmed, William Bowman, Dominique Devriese, Cătălin Hriţcu, Max Nex, Frank Piessens and Tamara Rezk for interesting discussions on the subject of this paper as well as the anonymous reviewers for useful feedback that has improved on the paper.

References

[1] M. Abadi. Protection in programming-language translations. In *Secure Internet programming*, pages 19–34. Springer-Verlag, 1999.

[2] M. Abadi, C. Fournet, and G. Gonthier. Secure communications processing for distributed languages. In *IEEE Symposium on Security and Privacy*, pages 74–88, 1999.

[3] M. Abadi, C. Fournet, and G. Gonthier. Authentication primitives and their compilation. In *POPL '00*, pages 302–315. ACM, 2000.

[4] M. Abadi, C. Fournet, and G. Gonthier. Secure implementation of channel abstractions. *Information and Computation*, 174:37–83, 2002.

[5] M. Abadi and G. Plotkin. On protection by layout randomization. In *CSF '10*, pages 337–351. IEEE, 2010.

[6] A. Ahmed. Verified Compilers for a Multi-Language World. In *SNAPL 2015*, volume 32, pages 15–31, Dagstuhl, Germany, 2015. Schloss Dagstuhl.

[7] A. Ahmed and M. Blume. Typed closure conversion preserves observational equivalence. *SIGPLAN Not.*, 43(9):157–168, 2008.

[8] A. Ahmed and M. Blume. An equivalence-preserving CPS translation via multi-language semantics. *SIGPLAN Not.*, 46(9):431–444, 2011.

[9] B. Alpern and F. B. Schneider. Defining liveness. Technical report, Ithaca, NY, USA, 1984.

[10] I. G. Baltopoulos and A. D. Gordon. Secure compilation of a multi-tier web language. In *TLDI '09*, pages 27–38. ACM, 2009.

[11] G. Barthe, T. Rezk, and A. Basu. Security types preserving compilation. *ELSEVIER Comlan*, 33:35–59, 2007.

[12] G. Barthe, T. Rezk, A. Russo, and A. Sabelfeld. Security of multithreaded programs by compilation. *ACM TISSEC*, 13:21:1–21:32, 2010.

[13] N. Benton and C.-K. Hur. Biorthogonality, step-indexing and compiler correctness. *SIGPLAN Not.*, 44(9):97–108, Aug. 2009.

[14] N. Benton and C.-k. Hur. Realizability and compositional compiler correctness for a polymorphic language. Technical report, MSR, 2010.

[15] N. Bielova and F. Massacci. Iterative enforcement by suppression: Towards practical enforcement theories. *J. Comput. Secur.*, 20(1):51–79, Jan. 2012.

[16] A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic. Reactive noninterference. In *CCS '09*, pages 79–90. ACM, 2009.

[17] W. J. Bowman and A. Ahmed. Noninterference for free. In *ICFP '15*, New York, NY, USA, 2015. ACM.

[18] M. Bugliesi and M. Giunti. Secure implementations of typed channel abstractions. In *POPL '07*, pages 251–262. ACM, 2007.

[19] M. R. Clarkson and F. B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, Sept. 2010.

[20] R. Corin, P.-M. Deniélou, C. Fournet, K. Bhargavan, and J. Leifer. A secure compiler for session abstractions. *Journal of Computer Security*, 16:573–636, 2008.

[21] D. Devriese, M. Patrignani, and F. Piessens. Secure Compilation by Approximate Back-Translation. In *POPL 2016*, 2016.

[22] U. Dhawan, N. Vasilakis, R. Rubin, S. Chiricescu, J. M. Smith, T. F. Knight, Jr., B. C. Pierce, and A. DeHon. Pump: A programmable unit for metadata processing. In *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '14, pages 8:1–8:8, New York, NY, USA, 2014. ACM.

[23] V. D'Silva, M. Payer, and D. X. Song. The correctness-security gap in compiler optimization. In *2015 IEEE S& P Workshops, SPW 2015*, pages 73–87, 2015.

[24] R. Focardi and R. Gorrieri. A classification of security properties for process algebras. *J. Comput. Secur.*, 3(1):5–33, Jan. 1995.

[25] C. Fournet, N. Swamy, J. Chen, P.-E. Dagand, P.-Y. Strub, and B. Livshits. Fully abstract compilation to javascript. In *POPL '13*, pages 371–384, New York, NY, USA, 2013. ACM.

[26] D. Gorla and U. Nestman. Full abstraction for expressiveness: History, myths and facts. *Math Struct Comp Science*, 2014.

[27] C.-K. Hur and D. Dreyer. A Kripke logical relation between ML and Assembly. *SIGPLAN Not.*, 46(1):133–146, Jan. 2011.

[28] R. Jagadeesan, C. Pitcher, J. Rathke, and J. Riely. Local memory via layout randomization. In *CSF '11*, pages 161–174, USA, 2011.

[29] A. Jeffrey and J. Rathke. Java Jr.: Fully abstract trace semantics for a core Java language. In *ESOP'05*, volume 3444 of *LNCS*, pages 423–438. Springer, 2005.

[30] Y. Juglaret, C. Hriţcu, A. Azevedo de Amorim, and B. C. Pierce. Beyond good and evil: Formalizing the security guarantees of compartmentalizing compilation. In *CSF*. IEEE Computer Society Press, July 2016.

[31] J. Laird. A fully abstract trace semantics for general references. In *Automata, Languages and Programming*, volume 4596 of *Lecture Notes in Computer Science*, pages 667–679. Springer Berlin Heidelberg, 2007.

[32] A. Larmuseau, M. Patrignani, and D. Clarke. A secure compiler for ML modules. In *APLAS 2015*, pages 29–48, 2015.

[33] P. Laud. Secure Implementation of Asynchronous Method Calls and Futures. In C. J. Mitchell and A. Tomlinson, editors, *Trusted Systems*, volume 7711 of *LNCS*, pages 25–47. Springer Berlin Heidelberg, 2012.

[34] J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *Int. J. Inf. Secur.*, 4(1-2):2–16, Feb. 2005.

[35] J. Matthews and R. B. Findler. Operational semantics for multi-language programs. *ACM Trans. Program. Lang. Syst.*, 31(3):12:1–12:44, Apr. 2009.

[36] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP '13*, pages 10:1–10:1. ACM, 2013.

[37] G. Neis, C.-K. Hur, J.-O. Kaiser, C. McLaughlin, D. Dreyer, and V. Vafeiadis. Pilsner: A compositionally verified compiler for a higher-order imperative language. In *ICFP 2015*, pages 166–178. ACM, 2015.

[38] M. New, W. J. Bowman, and A. Ahmed. Fully-abstract compilation via universal embedding. In *ICFP '16*, New York, NY, USA, 2016. ACM.

[39] J. Parrow. General conditions for full abstraction. *Math Struct Comp Science*, 2014.

[40] M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens. Secure Compilation to Protected Module Architectures. *ACM Trans. Program. Lang. Syst.*, 37(2):6:1–6:50, 2015.

[41] M. Patrignani and D. Clarke. Fully abstract trace semantics for protected module architectures. *ELSEVIER Comlan*, 42(0):22 – 45, 2015.

[42] M. Patrignani, D. Devriese, and F. Piessens. On Modular and Fully Abstract Compilation. In *CSF 2016*, 2016.

[43] F. Piessens, D. Devriese, J. T. Muhlberg, and R. Strackx. Security guarantees for the execution infrastructure of software applications. In *IEEE SecDev 2016*, 2016.

[44] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[45] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, Feb. 2000.

[46] S. Tse and S. Zdancewic. Translating dependency into parametricity. *SIGPLAN Not.*, 39:115–125, Sept. 2004.

[47] Y. Welsch and A. Poetzsch-Heffter. A fully abstract trace-based semantics for reasoning about backward compatibility of class libraries. *Science of Computer Programming*, -(0):–, 2013.

[48] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecture*, ISCA '14, pages 457–468, Piscataway, NJ, USA, 2014. IEEE Press.

[49] A. Zakinthinos and E. S. Lee. A general theory of security properties. In *IEEE S& P*, SP '97, pages 94–. IEEE Computer Society, 1997.

Appendix

The appendix contains proofs, supplementary material as well as a discussion on modular **TPC**.

A Proofs of Theorems Presented in the Paper

This section presents the proofs of all theorems and possible helper lemmas.

A.1 Proofs of Section 3 (Trace-Preserving Compilation)

Proof of Theorem 1 (Halting implies disregarding).

Proof. Take $\widehat{\surd} \equiv \surd$ and only traces that stutter on $\surd$. □

Proof of Theorem 2 (Source programs refine their compiled counterparts).

Proof. By definition. □

Proof of Theorem 3 (Equivalent programs have the same invalid traces).

Proof. This is proven by induction over n.

- *base* if n is 0 then $\mathsf{B}_{\mathbf{C}}$ is $\emptyset$, so this case is a trivial contradiction.
- *inductive* This is proven by contradiction.

  Suppose wlog $\exists\bar{\alpha}$ such that $\bar{\alpha} \in \mathsf{B}_{\mathbf{C_1}}$ but $\bar{\alpha} \notin \mathsf{B}_{\mathbf{C_2}}$.

  By $\bar{\alpha} \in \mathsf{B}_{\mathbf{C_1}}$ we have that $\bar{\alpha} \equiv \bar{\mathsf{m}}_1\alpha?\surd_{\mathsf{n}+1}\bar{\alpha}_2$ and $\forall\bar{\mathsf{m}}' \curlywedge \bar{\mathsf{m}}_1, \nexists\bar{\mathbf{m}}\boldsymbol{\alpha}? \in \mathtt{op}(\mathbf{TR}(\mathbf{C})).\bar{\mathbf{m}}\boldsymbol{\alpha}? \approx \bar{\mathsf{m}}'\alpha?$

  By $\bar{\alpha} \notin \mathsf{B}_{\mathbf{C_2}}$ we have that $\bar{\alpha} \equiv \bar{\mathsf{m}}'_1\alpha?\alpha!\bar{\alpha}'_2$ and $\exists\bar{\mathbf{m}}\boldsymbol{\alpha}? \in \mathtt{op}(\mathbf{TR}(\mathbf{C})).\bar{\mathbf{m}}\boldsymbol{\alpha}? \approx \bar{\mathsf{m}}'_1\alpha?$.

  We also know that $\bar{\mathsf{m}}'_1 \equiv \bar{\mathsf{m}}'$ because neither have $\surd$ s and they have a source-level counterpart.

  By definition we have that $\mathbf{TR}(\mathbf{C_1}) = \mathbf{TR}(\mathbf{C_2})$.

  This leads to the contradiction because we have both $\exists\bar{\mathbf{m}}\boldsymbol{\alpha}? \in \mathtt{op}(\mathbf{TR}(\mathbf{C})).\bar{\mathbf{m}}\boldsymbol{\alpha}? \approx \bar{\mathsf{m}}'\alpha?$ and $\nexists\bar{\mathbf{m}}\boldsymbol{\alpha}? \in \mathtt{op}(\mathbf{TR}(\mathbf{C})).\bar{\mathbf{m}}\boldsymbol{\alpha}? \approx \bar{\mathsf{m}}'\alpha?$.

□

A.2 Proofs of Section 4 (Trace-Preserving Compilation and Hyperproperty Preservation)

Proof of Theorem 5 (Safety preservation).

This proof is completely implied by the next one, but it clarifies the reasoning principle.

Proof. This proof proceeds by contradiction.

Suppose $\mathsf{TR}(\llbracket\mathbf{C}\rrbracket^{\mathcal{S}}_{\mathcal{T}}) \neq \mathsf{S}$, so $\exists\bar{\alpha}$ such that: $\bar{\alpha} \in \mathsf{TR}(\llbracket\mathbf{C}\rrbracket^{\mathcal{S}}_{\mathcal{T}})$ and $\bar{\alpha} \notin \mathsf{S}$. This second point implies that $\exists\overline{\mathsf{m}} \in \widehat{\overline{\mathsf{m}}}.\overline{\mathsf{m}} \leq \bar{\alpha}$.

By definition, there are two cases for $\bar{\alpha}$ it's either a trace with a source-level counterpart or an invalid trace:

1. $\bar{\alpha} \in \{\bar{\alpha} \mid \exists\bar{\boldsymbol{\alpha}} \in \mathbf{TR}(\mathbf{C}).\bar{\boldsymbol{\alpha}} \approx \bar{\alpha}\}$

   Because $\bar{\alpha} \notin \mathsf{S}$, we have that $\exists\overline{\mathsf{m}} \in \widehat{\overline{\mathsf{m}}}.\overline{\mathsf{m}} \leq \bar{\alpha}$.

   By definition, $\overline{\mathsf{m}}$ can be of two kinds:

   (a) $\exists\overline{\mathbf{m}} \in \widehat{\overline{\mathbf{m}}}, \overline{\mathbf{m}} \approx \overline{\mathsf{m}}$.

   Because $\overline{\mathsf{m}} \leq \bar{\alpha}$ and $\bar{\boldsymbol{\alpha}} \approx \bar{\alpha}$, we have that $\overline{\mathbf{m}} \leq \bar{\boldsymbol{\alpha}}$.

   Since $\bar{\boldsymbol{\alpha}} \in \mathbf{TR}(\mathbf{C})$, by functionality of Definition 4 we have that $\nexists\overline{\mathbf{m}} \in \widehat{\overline{\mathbf{m}}}.\overline{\mathbf{m}} \leq \bar{\boldsymbol{\alpha}}$.

   So the absurdum is reached: $\exists$ and $\nexists\overline{\mathbf{m}} \in \widehat{\overline{\mathbf{m}}}.\overline{\mathbf{m}} \leq \bar{\boldsymbol{\alpha}}$.

   (b) $\overline{\mathsf{m}} \equiv \overline{\mathsf{m}}''\alpha?\alpha!$ $\exists\overline{\mathbf{m}} \approx \overline{\mathsf{m}}$ and $\nexists\boldsymbol{\alpha}? \approx \alpha?$ and $\alpha! \neq \surd$

   This cannot be, since $\overline{\mathsf{m}} \leq \bar{\alpha}$ and $\bar{\boldsymbol{\alpha}} \approx \bar{\alpha}$ and $\boldsymbol{\alpha}? \approx \alpha?$ contradicts Rule Relate-trace.

2. $\bar{\alpha} \in \{\bar{\alpha} \mid \mathtt{obs}(\bar{\alpha}) = \bigcup_{n\in\mathbb{N}} \mathtt{int}_{\mathtt{n}}(\mathbf{C})\}$

   The following cases arise:

   (a) $\overline{\mathsf{m}} \equiv \bar{\mathsf{m}}_1$ and $\nexists\surd_{\mathsf{i}} \in \bar{\mathsf{m}}_1$.

   This case is just like Item 1a.

   (b) $\overline{\mathsf{m}} \equiv \bar{\mathsf{m}}_1\alpha?\surd_{\mathsf{i}}$ for some i and $\nexists\surd_{\mathsf{j}} \in \bar{\mathsf{m}}_1$.

   Again, since $\widehat{\overline{\mathbf{m}}} \approx \widehat{\overline{\mathsf{m}}}$, we have that $\overline{\mathsf{m}}$ can be of two kinds:

   i. $\exists\overline{\mathbf{m}} \in \widehat{\overline{\mathbf{m}}}, \overline{\mathbf{m}} \approx \overline{\mathsf{m}}$.
   This is an absurdum because $\nexists\boldsymbol{\alpha} \approx \surd_{\mathsf{i}}$.

   ii. $\overline{\mathsf{m}} \equiv \overline{\mathsf{m}}''\alpha?\alpha!$ $\exists\overline{\mathbf{m}} \approx \overline{\mathsf{m}}$ and $\nexists\boldsymbol{\alpha}? \approx \alpha?$ and $\alpha! \neq \surd$
   The absurdum here is that $\alpha! \neq \surd$.

   (c) $\overline{\mathsf{m}} \equiv \bar{\mathsf{m}}_1\alpha?\surd_{\mathsf{i}}\bar{\alpha}_2$.

   In this case we can restrict ourselves to the prefix $\overline{\mathsf{m}} \equiv \bar{\mathsf{m}}_1\alpha?\surd_{\mathsf{i}}$ and use the same reasoning as in Item 2b.

□

Proof of Theorem 6 (Hypersafety preservation).

Proof. This proof proceeds by contradiction.

Suppose $\mathsf{TR}(\llbracket \mathbf{C} \rrbracket^{\mathcal{S}}_{\mathcal{T}}) \notin \mathbb{S}$, so $\exists \bar{\alpha}$ such that: $\bar{\alpha} \in \mathsf{TR}(\llbracket \mathbf{C} \rrbracket^{\mathcal{S}}_{\mathcal{T}})$ and $\nexists \widehat{\bar{\alpha}} \in \mathbb{S}. \bar{\alpha} \in \widehat{\bar{\alpha}}$.

By definition, there are two cases for $\bar{\alpha}$ it's either a trace with a source-level counterpart or an invalid trace:

1. $\bar{\alpha} \in \{\bar{\alpha} \mid \exists \bar{\boldsymbol{\alpha}} \in \mathbf{TR}(\mathbf{C}). \bar{\boldsymbol{\alpha}} \approx \bar{\alpha}\}$

   Because $\nexists \widehat{\bar{\alpha}} \in \mathbb{S}. \bar{\alpha} \in \widehat{\bar{\alpha}}$, we have that $\exists \widehat{\overline{\mathsf{m}}} \in \mathbb{M}. \exists \overline{\mathsf{m}} \in \widehat{\overline{\mathsf{m}}}. \overline{\mathsf{m}} \leq \bar{\alpha}$.

   By definition, $\overline{\mathsf{m}}$ can be of two kinds:

   (a) $\exists \widehat{\overline{\mathbf{m}}} \in \mathbb{M}, \widehat{\overline{\mathsf{m}}} \approxeq \widehat{\overline{\mathbf{m}}}$, so $\exists \overline{\mathbf{m}} \in \widehat{\overline{\mathbf{m}}}.\ \overline{\mathbf{m}} \approx \overline{\mathsf{m}}$.

   Since $\overline{\mathsf{m}} \leq \bar{\alpha}$ and $\bar{\boldsymbol{\alpha}} \approx \bar{\alpha}$, we have that $\overline{\mathbf{m}} \leq \bar{\boldsymbol{\alpha}}$

   Since $\bar{\boldsymbol{\alpha}} \in \mathbf{TR}(\mathbf{C})$, by functionality of Definition 4 we have that $\nexists \widehat{\overline{\mathbf{m}}} \in \mathbb{M}. \exists \overline{\mathbf{m}} \in \widehat{\overline{\mathbf{m}}}. \overline{\mathbf{m}} \leq \bar{\boldsymbol{\alpha}}$.

   So the absurdum is reached: $\exists$ and $\nexists \overline{\mathbf{m}} \in \widehat{\overline{\mathbf{m}}}. \overline{\mathbf{m}} \leq \bar{\boldsymbol{\alpha}}$.

   (b) $\{\{\bar{\alpha}\alpha?\alpha!\} \mid \exists \widehat{\bar{\boldsymbol{\alpha}}} \in \mathbb{M}. \exists \bar{\boldsymbol{\alpha}} \in \widehat{\bar{\boldsymbol{\alpha}}}. \bar{\boldsymbol{\alpha}} \approx \bar{\alpha}$ and $\nexists \widehat{\bar{\boldsymbol{\alpha}}'} \in \mathbb{M}. \exists \bar{\boldsymbol{\alpha}}'\boldsymbol{\alpha}? \in \widehat{\bar{\boldsymbol{\alpha}}'}. \bar{\boldsymbol{\alpha}}'\boldsymbol{\alpha}? \approx \bar{\alpha}\alpha?$ and $\alpha! \neq \surd\}$

   This contradicts the assumption $\exists \bar{\boldsymbol{\alpha}} \in \mathbf{TR}(\mathbf{C}). \bar{\boldsymbol{\alpha}} \approx \bar{\alpha}$.

2. $\bar{\alpha} \in \{\bar{\alpha} \mid \mathtt{obs}(\bar{\alpha}) = \bigcup_{n \in \mathbb{N}} \mathtt{int_n}(\mathbf{C})\}$

   The following cases arise:

   (a) $\overline{\mathsf{m}} \equiv \bar{\mathsf{m}}_1$ and $\nexists \surd_{\mathsf{i}} \in \bar{\mathsf{m}}_1$.

   This case is just like Item 1a.

   (b) $\overline{\mathsf{m}} \equiv \bar{\mathsf{m}}_1 \alpha? \surd_{\mathsf{i}}$ for some i and $\nexists \surd_{\mathsf{j}} \in \bar{\mathsf{m}}_1$.

   By definition we have that $\exists \widehat{\overline{\mathsf{m}}} \in \mathbb{M}.\ \exists \overline{\mathsf{m}} \in \widehat{\overline{\mathsf{m}}}$ and $\widehat{\overline{\mathsf{m}}}$ can be of two kinds:

   i. $\exists \widehat{\overline{\mathbf{m}}} \in \mathbb{M}. \exists \overline{\mathbf{m}} \in \widehat{\overline{\mathbf{m}}}, \overline{\mathbf{m}} \approx \overline{\mathsf{m}}$.

   This is an absurdum because $\nexists \boldsymbol{\alpha} \approx \surd_{\mathsf{i}}$.

   ii. $\overline{\mathsf{m}} \equiv \overline{\mathsf{m}}''\alpha?\alpha!\ \exists \widehat{\overline{\mathbf{m}}} \in \mathbb{M}. \exists \overline{\mathbf{m}} \in \widehat{\overline{\mathbf{m}}}. \overline{\mathbf{m}} \approx \overline{\mathsf{m}}$ and $\nexists \boldsymbol{\alpha}? \approx \alpha?$ and $\alpha! \neq \surd$

   The absurdum here is that $\alpha! \neq \surd$.

   (c) $\overline{\mathsf{m}} \equiv \bar{\mathsf{m}}_1 \alpha? \surd_{\mathsf{i}} \bar{\alpha}_2$.

   In this case we can restrict ourselves to the prefix $\overline{\mathsf{m}} \equiv \bar{\mathsf{m}}_1 \alpha? \surd_{\mathsf{i}}$ and use the same reasoning as in Item 2b.

□

Proof of Theorem 7 (Non-interference is preserved).

Proof. This proof proceeds by contradiction.

Suppose (1) $\widehat{\bar{\alpha}} \in \mathbb{S}$ and (2) $\widehat{\bar{\alpha}} \notin \mathbb{NI}$.

By (2) we know that $\exists \bar{\alpha}_1, \bar{\alpha}_2 \in \widehat{\bar{\alpha}}$ such that $\bar{\alpha}_1|_I \overset{\mathrm{T}}{=}_L \bar{\alpha}_2|_I$ and $\bar{\alpha}_1|_O \overset{\mathrm{T}}{\neq}_L \bar{\alpha}_2|_O$.

By (1) we know that $\nexists \widehat{\overline{\mathsf{m}}} \in \mathbb{M}$ such that (3) $\widehat{\overline{\mathsf{m}}} \leq \widehat{\bar{\alpha}}$.

There are two cases based on $\widehat{\overline{\mathsf{m}}}$:

- $\widehat{\overline{\mathsf{m}}} \in \mathsf{A}$ where $\mathsf{A} = \{\widehat{\overline{\mathsf{m}}} \mid \exists \widehat{\overline{\boldsymbol{\alpha}}} \in \mathbb{M}, \widehat{\overline{\alpha}} ≋ \widehat{\overline{\boldsymbol{\alpha}}}\}$

  By Definition 15, $\mathsf{A} = \{\widehat{\overline{\mathsf{m}}} \mid \forall \overline{\mathsf{m}_1}, \overline{\mathsf{m}_2} \in \widehat{\overline{\mathsf{m}}}.\overline{\mathsf{m}_1}|_I \overset{\mathrm{T}}{=}_L \overline{\mathsf{m}_2}|_I$ and $\overline{\mathsf{m}_1}|_O \overset{\mathrm{T}}{\neq}_L \overline{\mathsf{m}_2}|_O\}$

  By (3) we conclude that $\forall \overline{\alpha}_1, \overline{\alpha}_2 \in \widehat{\overline{\alpha}}$, the following holds $\overline{\alpha}_1|_I \overset{\mathrm{T}}{=}_L \overline{\alpha}_2|_I$ and $\overline{\alpha}_1|_O \overset{\mathrm{T}}{=}_L \overline{\alpha}_2|_O$.

  This contradicts (2).

- $\widehat{\overline{\mathsf{m}}} \in \mathsf{B}$ where $\mathsf{B} = \{\{\overline{\mathsf{m}}\alpha?\alpha!\} \mid \exists \widehat{\overline{\boldsymbol{\alpha}}} \in \mathbb{M}.\exists \overline{\mathbf{m}} \in \widehat{\overline{\boldsymbol{\alpha}}}.\overline{\mathbf{m}} \approx \overline{\mathsf{m}}$ and $\nexists \widehat{\overline{\boldsymbol{\alpha}}'} \in \mathbb{M}.\exists \overline{\mathbf{m}}'\boldsymbol{\alpha}? \in \widehat{\overline{\boldsymbol{\alpha}}'}.\overline{\mathbf{m}}'\boldsymbol{\alpha}? \approx \overline{\mathsf{m}}\alpha?$ and $\alpha! \neq \surd_{i+1}\}$ where $\|\overline{\mathsf{m}}|_O \cap \widehat{\surd}\| = i$

  Assume wlog that $\overline{\alpha}_1 \equiv \overline{\mathsf{m}}'_1\alpha_1?\alpha_1!\overline{\alpha}''_1$ and $\overline{\alpha}_2 \equiv \overline{\mathsf{m}}'_2\alpha_2?\alpha_2!\overline{\alpha}''_2$.

  By hypothesis we get $\overline{\mathsf{m}}'_1\alpha_1? \overset{\mathrm{T}}{=}_L \overline{\mathsf{m}}'_2\alpha_2?$ and (4) $\alpha_1! \overset{\mathrm{T}}{\neq}_L \alpha_2!$.

  By (3) we have that $\alpha_1! \equiv \surd_i$ where $\|\overline{\mathsf{m}}'_1|_O \cap \widehat{\surd}\| = i$.

  Analogously, by (3) we have that $\alpha_2! \equiv \surd_i$ where $\|\overline{\mathsf{m}}'_2|_O \cap \widehat{\surd}\| = i$.

  So, $\alpha_1! \equiv \surd_i \equiv \alpha_2!$, which contradicts (4).

□

Proof of Theorem 8 (Liveness preservation).

Proof. This proof proceeds by contradiction.

Suppose there is a trace $\overline{\alpha}$ at that is in $\mathsf{TR}(\llbracket \mathbf{C} \rrbracket^{\mathcal{S}}_{\mathcal{T}})$ but not in $\widehat{\overline{\mathsf{L}}}$.
By $\widehat{\overline{\mathbf{L}}} \overset{\mathrm{LP}}{\approx} \widehat{\overline{\mathsf{L}}}$ we know that any trace in $\widehat{\overline{\mathsf{L}}}$:

- is $\mathsf{fair}()$. By contradiction, no input $\alpha?$ in $\overline{\alpha}$ after a $\surd$ should be related to a source input $\boldsymbol{\alpha}?$.

  We proceed by induction on $\overline{\alpha}$.

  In the base case we know that $\exists \overline{\boldsymbol{\alpha}}.\overline{\boldsymbol{\alpha}} \approx \overline{\alpha}$, so $\overline{\alpha}$ has an input $\alpha?$ such that $\exists \boldsymbol{\alpha}? \in \overline{\boldsymbol{\alpha}}.\ \boldsymbol{\alpha}? \approx \boldsymbol{\alpha}?$ and the contradiction is reached.

  The inductive case follows by IH.

- once stripped of ticks, has a related trace in $\widehat{\overline{\mathbf{L}}}$.

  By Definition 8 we know that once we strip $\overline{\alpha}$ of ticks, $\exists \overline{\boldsymbol{\alpha}} \in \mathbf{TR}(\mathbf{C}).\overline{\boldsymbol{\alpha}} \approx \overline{\alpha}$.

  By $\mathbf{TR}(\mathbf{C}) \subseteq \widehat{\overline{\mathbf{L}}}$, we have that $\overline{\boldsymbol{\alpha}} \in \widehat{\overline{\mathbf{L}}}$, so the contradiction is reached. □

A.3 Proofs (and Examples) of Section 5 (Trace-Preserving and Fully Abstract Compilation)

Definition 24 (Compiler full abstraction [1]). $\llbracket \cdot \rrbracket^{\mathcal{S}}_{\mathcal{T}} \in \mathit{FA} \overset{\mathsf{def}}{=} \forall \mathbf{C_1}, \mathbf{C_2}.\ \mathbf{C_1} \simeq_{ctx} \mathbf{C_2} \iff \llbracket \mathbf{C_1} \rrbracket^{\mathcal{S}}_{\mathcal{T}} \simeq_{ctx} \llbracket \mathbf{C_2} \rrbracket^{\mathcal{S}}_{\mathcal{T}}$.

Definition 24 and Definition 6 are equivalent as long as Assumption 2 and Assumption 1 hold.

Proof of Theorem 9 (*TP* implies *FA*).

Proof. $\Rightarrow$ $\mathbf{C_1} \overset{\mathrm{T}}{=} \mathbf{C_2} \Rightarrow \llbracket \mathbf{C_1} \rrbracket^{\mathcal{S}}_{\mathcal{T}} \overset{\mathrm{T}}{=} \llbracket \mathbf{C_2} \rrbracket^{\mathcal{S}}_{\mathcal{T}}$

By Definition 3 $\mathbf{TR}(\mathbf{C_1}) = \mathbf{TR}(\mathbf{C_2})$

By Theorem 3 $\mathsf{B}_{\mathbf{C_1}} = \mathsf{B}_{\mathbf{C_2}}$.

By Definition 15 $\mathsf{G}_{\mathbf{C_1}} \approxeq \mathbf{TR}(\mathbf{C_1})$ and $\mathsf{G}_{\mathbf{C_2}} \approxeq \mathbf{TR}(\mathbf{C_2})$.

So $\mathsf{G}_{\mathbf{C_1}} + \mathsf{B}_{\mathbf{C_1}} = \mathsf{G}_{\mathbf{C_2}} + \mathsf{B}_{\mathbf{C_2}}$.

By Definition 1, $\mathsf{TR}(\llbracket \mathbf{C_1} \rrbracket^{\mathcal{S}}_{\mathcal{T}}) = \mathsf{TR}(\llbracket \mathbf{C_2} \rrbracket^{\mathcal{S}}_{\mathcal{T}})$.

By Definition 3 $\llbracket \mathbf{C_1} \rrbracket^{\mathcal{S}}_{\mathcal{T}} \overset{\mathrm{T}}{=} \llbracket \mathbf{C_2} \rrbracket^{\mathcal{S}}_{\mathcal{T}}$.

$\Leftarrow$ $\llbracket \mathbf{C_1} \rrbracket^{\mathcal{S}}_{\mathcal{T}} \overset{\mathrm{T}}{=} \llbracket \mathbf{C_2} \rrbracket^{\mathcal{S}}_{\mathcal{T}} \Rightarrow \mathbf{C_1} \overset{\mathrm{T}}{=} \mathbf{C_2}$

By Definition 3 $\mathsf{TR}(\llbracket \mathbf{C_1} \rrbracket^{\mathcal{S}}_{\mathcal{T}}) = \mathsf{TR}(\llbracket \mathbf{C_2} \rrbracket^{\mathcal{S}}_{\mathcal{T}})$.

By Definition 1, $\mathsf{G}_{\mathbf{C_1}} + \mathsf{B}_{\mathbf{C_1}} = \mathsf{G}_{\mathbf{C_2}} + \mathsf{B}_{\mathbf{C_2}}$.

By totality of Definition 4, $\mathsf{B}_{\mathbf{C_1}} = \mathsf{B}_{\mathbf{C_2}}$.

So $\mathsf{G}_{\mathbf{C_1}} = \mathsf{G}_{\mathbf{C_2}}$.

By Definition 15 $\mathsf{G}_{\mathbf{C_1}} \approxeq \mathbf{TR}(\mathbf{C_1})$ and $\mathsf{G}_{\mathbf{C_2}} \approxeq \mathbf{TR}(\mathbf{C_2})$.

So $\mathbf{TR}(\mathbf{C_1}) = \mathbf{TR}(\mathbf{C_2})$.

By Definition 3 $\mathbf{C_1} \overset{\mathrm{T}}{=} \mathbf{C_2}$ □

Theorem 15 (Modular **TPC** implies modular **FAC**)**.** $\llbracket \cdot \rrbracket^{\mathcal{S}}_{\mathcal{T}} \in \mathit{MTP} \Rightarrow \llbracket \cdot \rrbracket^{\mathcal{S}}_{\mathcal{T}} \in \mathit{MFA}$.

Proof. The proof is completely analogous to the single-module case. □

Assumption 3 ($\llbracket \cdot \rrbracket^{\lambda^{B}}_{\lambda^{\mathbb{N}}}$ is correct)**.** $\llbracket \cdot \rrbracket^{\lambda^{B}}_{\lambda^{\mathbb{N}}} \in \mathit{CC}$

Theorem 16 ($\llbracket \cdot \rrbracket^{\lambda^{B}}_{\lambda^{\mathbb{N}}}$ is fully abstract)**.** $\llbracket \cdot \rrbracket^{\lambda^{B}}_{\lambda^{\mathbb{N}}} \in \mathit{FA}$

Proof. $\forall \mathbf{C_1}, \mathbf{C_2}, \mathbf{C_1} \overset{\mathrm{T}}{=} \mathbf{C_2} \iff \llbracket \mathbf{C_1} \rrbracket^{\lambda^{B}}_{\lambda^{\mathbb{N}}} \overset{\mathrm{T}}{=} \llbracket \mathbf{C_2} \rrbracket^{\lambda^{B}}_{\lambda^{\mathbb{N}}}$

Let $\mathbf{C_1} = \mathbf{f}(\mathbf{x}) = \mathbf{t_1}$ and $\mathbf{C_2} = \mathbf{f}(\mathbf{x}) = \mathbf{t_2}$.

By definition we know that $\llbracket \mathbf{C_1} \rrbracket^{\lambda^{B}}_{\lambda^{\mathbb{N}}} = \mathsf{f}(\mathsf{x}) = \lambda \mathsf{x}.(\llbracket \mathbf{t_1} \rrbracket^{\lambda^{B}}_{\lambda^{\mathbb{N}}})\ \mathtt{min}(\mathsf{x}, 1)$.

Analogously $\llbracket \mathbf{C_2} \rrbracket^{\lambda^{B}}_{\lambda^{\mathbb{N}}} = \mathsf{f}(\mathsf{x}) = \lambda \mathsf{x}.(\llbracket \mathbf{t_2} \rrbracket^{\lambda^{B}}_{\lambda^{\mathbb{N}}})\ \mathtt{min}(\mathsf{x}, 1)$.

So any trace of the compiled components has inputs of the form $\mathsf{callf}(\mathsf{v})?$ for $\mathsf{v} \in \mathbb{N}$.

$\mathbf{C_1} \overset{\mathbf{T}}{=} \mathbf{C_2} \Rightarrow \llbracket \mathbf{C_1} \rrbracket^{\lambda^{B}}_{\lambda^{\mathbb{N}}} \overset{\mathbf{T}}{=} \llbracket \mathbf{C_2} \rrbracket^{\lambda^{B}}_{\lambda^{\mathbb{N}}}$

We state the contrapositive: $\llbracket \mathbf{C_1} \rrbracket^{\lambda^{B}}_{\lambda^{\mathbb{N}}} \overset{\not\mathrm{T}}{\neq} \llbracket \mathbf{C_2} \rrbracket^{\lambda^{B}}_{\lambda^{\mathbb{N}}} \Rightarrow \mathbf{C_1} \overset{\not\mathrm{T}}{\neq} \mathbf{C_2}$

By definition, wlog, $\exists \bar{\alpha} \equiv \bar{\alpha}'\alpha?\alpha!$ such that $\bar{\alpha} \in \mathsf{TR}(\llbracket \mathbf{C_1} \rrbracket^{\lambda^{B}}_{\lambda^{\mathbb{N}}})$ and $\bar{\alpha} \notin \mathsf{TR}(\llbracket \mathbf{C_2} \rrbracket^{\lambda^{B}}_{\lambda^{\mathbb{N}}})$ and $\bar{\alpha}' \in \mathsf{TR}(\llbracket \mathbf{C_1} \rrbracket^{\lambda^{B}}_{\lambda^{\mathbb{N}}})$ and $\bar{\alpha}' \in \mathsf{TR}(\llbracket \mathbf{C_2} \rrbracket^{\lambda^{B}}_{\lambda^{\mathbb{N}}})$.

By input totality we can say that $\exists \alpha'!.\ \bar{\alpha}'\alpha?\alpha'! \in \mathsf{TR}(\llbracket \mathbf{C_2} \rrbracket^{\lambda^{B}}_{\lambda^{\mathbb{N}}})$.

We proceed by induction on $\bar{\alpha}'$ and then case analysis on $\alpha?$

By IH we get $\bar{\boldsymbol{\alpha}}' \approx \bar{\alpha}'$.

$\alpha? \equiv \mathsf{callf}(0)?$

By definition we get $\boldsymbol{\alpha}? \equiv \mathbf{callf}(\mathbf{false})?$ such that $\boldsymbol{\alpha}? \approx \alpha?$.

By Assumption 3 we get $\boldsymbol{\alpha}! \approx \alpha!$ and $\boldsymbol{\alpha}'! \approx \alpha'!$ and $\boldsymbol{\alpha}! \neq \boldsymbol{\alpha}'!$.

So we have $\bar{\boldsymbol{\alpha}} \equiv \bar{\boldsymbol{\alpha}}'\boldsymbol{\alpha}?\boldsymbol{\alpha}!$ such that $\bar{\boldsymbol{\alpha}} \in \mathbf{TR}(\mathbf{C_1})$ but $\bar{\boldsymbol{\alpha}} \notin \mathbf{TR}(\mathbf{C_2})$.

So $\mathbf{C_1} \overset{\mathbf{T}}{\not\equiv} \mathbf{C_2}$.

$\alpha? \equiv \mathsf{callf}(1)?$

Analogous as the case above.

$\alpha? \equiv \mathsf{callf}(\mathsf{n})?$ **where** $\mathsf{n} > 1$

The additional $\mathtt{min}(\cdot)$ turns the n into a 1, so this follows from the case above.

$\mathbf{C_1} \overset{\mathbf{T}}{=} \mathbf{C_2} \Leftarrow [\![\mathbf{C_1}]\!]^{\lambda^B}_{\lambda^{\mathbb{N}}} \overset{\mathbf{T}}{=} [\![\mathbf{C_2}]\!]^{\lambda^B}_{\lambda^{\mathbb{N}}}$

This direction is implied by Assumption 3.

□

A.3.1 Proof of Theorem 11 (Correctness and fail-safe behaviour imply trace-preservation)

Proof. This proof proceeds by contradiction.

Assume $[\![\cdot]\!]^{\mathcal{S}}_{\mathcal{T}} \in FA$, $[\![\cdot]\!]^{\mathcal{S}}_{\mathcal{T}} \in CC$ and $[\![\cdot]\!]^{\mathcal{S}}_{\mathcal{T}} \in FSB$ but $[\![\cdot]\!]^{\mathcal{S}}_{\mathcal{T}} \notin TP^H$.

By Definition 8: $\forall \mathbf{C}.\ \nexists \bar{\boldsymbol{\alpha}} \in \mathbf{TR}(\mathbf{C}).\ \bar{\boldsymbol{\alpha}} \approx \bar{\alpha}^f$ and $\bar{\alpha}^f \notin \{\bar{\mathsf{m}}\alpha?\surd\bar{\alpha}' \mid \exists \bar{\mathbf{m}} \in \mathtt{obs}(\mathbf{TR}(\mathbf{C})).\bar{\mathbf{m}} \approx \bar{\mathsf{m}}$ and $\nexists \bar{\mathbf{m}}\boldsymbol{\alpha}? \in \mathtt{op}(\mathbf{TR}(\mathbf{C})).\bar{\mathbf{m}}\boldsymbol{\alpha}? \approx \bar{\mathsf{m}}\alpha?$ and $\forall \alpha' \in \bar{\alpha}'|_O, \alpha' \equiv \surd\}$.

- The first conjunct cannot be possible since $[\![\cdot]\!]^{\mathcal{S}}_{\mathcal{T}} \in CC$.
- By definition of FSB, the second conjunct is also not possible. In fact $\bar{\alpha}^f \equiv \bar{\mathsf{m}}_1\alpha?\surd\bar{\alpha}_2$ and $\exists \bar{\mathbf{m}}_1 \in \mathtt{obs}(\mathbf{TR}(\mathbf{C})).\ \bar{\mathbf{m}}_1 \approx \bar{\mathsf{m}}_1$ and $\nexists \boldsymbol{\alpha}? \approx \alpha?$ and $\nexists \surd \in \bar{\mathsf{m}}_1$ and $\bar{\alpha}_2|_O = \surd$ but this contradicts the second conjunct. □

A.4 Proofs of Section 6.3 (TPC for an Existing Fully Abstract Compiler)

Additional examples for Theorem 13 ($[\![\cdot]\!]^{\lambda^{\tau}}_{\lambda^{\mathsf{u}}}$ has fail-safe behaviour).

Example 8 (Reduction to λ-term that will reduce to wrong)**.** Consider the term $\mathbf{f} = \lambda\mathbf{x} : \mathtt{Unit}.\ \mathbf{unit}$ that is compiled to $\mathsf{f} = \lambda\mathsf{y}.\ \lambda\mathsf{x}.\ \lambda\mathsf{z}.\ \mathsf{z}\ (\mathsf{y}\ (\lambda\mathsf{w}.\ (\mathsf{w}; \mathsf{unit})\ \mathsf{x}))\ \lambda\mathsf{x}.\ \mathsf{unit}$. Consider the following non-well-behaving context for f: $\mathbb{C}_{\mathsf{f}} = [\cdot]\ \mathsf{true}$. Once f is plugged into $\mathbb{C}_{\mathsf{f}}$, the resulting term performs the following reductions:

$$
\begin{aligned}
&(\lambda \mathsf{y}.\, \lambda \mathsf{x}.\, \lambda \mathsf{z}.\, \mathsf{z}\ (\mathsf{y}\ (\lambda \mathsf{w}.\, (\mathsf{w}; \mathsf{unit})\ \mathsf{x}))\ \lambda \mathsf{x}.\, \mathsf{unit})\ \mathsf{true} \hookrightarrow\\
&(\lambda \mathsf{x}.\, \lambda \mathsf{z}.\, \mathsf{z}\ (\lambda \mathsf{x}.\, \mathsf{unit}\ (\lambda \mathsf{w}.\, (\mathsf{w}; \mathsf{unit})\ \mathsf{x})))\ \mathsf{true} \hookrightarrow\\
&\lambda \mathsf{z}.\, \mathsf{z}\ (\lambda \mathsf{x}.\, \mathsf{unit}\ (\lambda \mathsf{w}.\, (\mathsf{w}; \mathsf{unit})\ \mathsf{true})) \hookrightarrow\\
&\lambda \mathsf{x}.\, \mathsf{unit}\ (\lambda \mathsf{w}.\, (\mathsf{w}; \mathsf{unit})\ \mathsf{true}) \hookrightarrow\\
&\lambda \mathsf{x}.\, \mathsf{unit}\ (\mathsf{true}; \mathsf{unit}) \hookrightarrow\\
&\lambda \mathsf{x}.\, \mathsf{unit}\ (\mathsf{wrong}) \hookrightarrow \mathsf{wrong}
\end{aligned}
$$

⊡

Example 9 (Reduction to λ-term that will reduce to wrong)**.** Consider the term $\mathbf{f} = \lambda\mathbf{x} : \texttt{Unit} \to \texttt{Unit}.\, \lambda\mathbf{o} : \texttt{Unit}.\, \mathbf{unit}$ of type $(\texttt{Unit} \to \texttt{Unit}) \to (\texttt{Unit} \to \texttt{Unit})$ that is compiled to the following (where f is $\lambda \mathsf{x}.\, \lambda \mathsf{o}.\, \mathsf{unit}$ for space reasons): $\mathsf{f} = \lambda \mathsf{y}.\, \lambda \mathsf{x}. \mathsf{protect}_{\texttt{Unit}\to\texttt{Unit}}\ (\mathsf{y}\ (\mathsf{confine}_{\texttt{Unit}\to\texttt{Unit}}\ \mathsf{x}))\ \mathsf{f}$. Consider the following non-well-behaving context for f: $\mathbb{C}_{\mathsf{f}} = [\cdot]\ \mathsf{true}$. Once f is plugged into $\mathbb{C}_{\mathsf{f}}$, the resulting term performs the following reductions:

$$
\begin{aligned}
&\lambda \mathsf{y}.\, \lambda \mathsf{x}.\, \mathsf{protect}_{\texttt{Unit}\to\texttt{Unit}}\ (\mathsf{y}\ (\mathsf{confine}_{\texttt{Unit}\to\texttt{Unit}}\ \mathsf{x}))\ \mathsf{f}\ \mathsf{true} \hookrightarrow\\
&\lambda \mathsf{x}.\, \mathsf{protect}_{\texttt{Unit}\to\texttt{Unit}}\ (\mathsf{f}\ (\mathsf{confine}_{\texttt{Unit}\to\texttt{Unit}}\ \mathsf{x}))\ \mathsf{true} \hookrightarrow\\
&\mathsf{protect}_{\texttt{Unit}\to\texttt{Unit}}\ (\mathsf{f}\ (\mathsf{confine}_{\texttt{Unit}\to\texttt{Unit}}\ \mathsf{true})) \hookrightarrow\\
&(\lambda \mathsf{y}.\, \lambda \mathsf{x}. \lambda \mathsf{z}.\, \mathsf{z}\ (\mathsf{y}\ (\mathsf{confine}_{\texttt{Unit}}\ \mathsf{x})))\\
&\qquad (\mathsf{f}\ (\mathsf{confine}_{\texttt{Unit}\to\texttt{Unit}}\ \mathsf{true})) \hookrightarrow\\
&(\lambda \mathsf{x}. \lambda \mathsf{z}.\, \mathsf{z}\ ((\mathsf{f}\ (\mathsf{confine}_{\texttt{Unit}\to\texttt{Unit}}\ \mathsf{true}))\ (\mathsf{confine}_{\texttt{Unit}}\ \mathsf{x})))
\end{aligned}
$$

However, if we were to supply an argument (called a) to this λ, the reduction would proceed as follows.

$$\begin{array}{l}
(\lambda \mathsf{x}.\lambda \mathsf{z}.\,\mathsf{z}\ ((\mathsf{f}\ (\mathsf{confine}_{\texttt{Unit}\to\texttt{Unit}}\ \mathsf{true}))\ (\mathsf{confine}_{\texttt{Unit}}\ \mathsf{x})))\ \mathsf{a} \hookrightarrow \\
(\lambda \mathsf{z}.\,\mathsf{z}\ ((\mathsf{f}\ (\mathsf{confine}_{\texttt{Unit}\to\texttt{Unit}}\ \mathsf{true}))\ (\mathsf{confine}_{\texttt{Unit}}\ \mathsf{a}))) \hookrightarrow \\
(((\mathsf{f}\ (\mathsf{confine}_{\texttt{Unit}\to\texttt{Unit}}\ \mathsf{true}))\ (\mathsf{confine}_{\texttt{Unit}}\ \mathsf{a}))) \hookrightarrow \\
(((\mathsf{f}\ (\mathsf{confine}_{\texttt{Unit}\to\texttt{Unit}}\ \mathsf{true}))\ (\mathsf{confine}_{\texttt{Unit}}\ \mathsf{a}))) \hookrightarrow \\
(((\mathsf{f}\ (\mathsf{confine}_{\texttt{Unit}\to\texttt{Unit}}\ \mathsf{true}))\ (\cdots))) \hookrightarrow \\
(((\mathsf{f}\ (\lambda \mathsf{y}.\,\lambda \mathsf{x}.\,\mathsf{confine}_{\texttt{Unit}}\ (\mathsf{y}\ (\mathsf{protect}_{\texttt{Unit}}\ \mathsf{x}))\ \mathsf{true}))\ (\cdots))) \hookrightarrow \\
(((\mathsf{f}\ (\lambda \mathsf{x}.\,\mathsf{confine}_{\texttt{Unit}}\ (\mathsf{true}\ (\mathsf{protect}_{\texttt{Unit}}\ \mathsf{x}))))\ (\cdots))) \hookrightarrow \\
(((\mathsf{f}\ (\lambda \mathsf{x}.\,\mathsf{confine}_{\texttt{Unit}}\ (\mathsf{true}\ (\mathsf{protect}_{\texttt{Unit}}\ \mathsf{x}))))\ (\mathsf{confine}_{\texttt{Unit}}\ \mathsf{a}))) \hookrightarrow \\
(((\mathsf{f}\ (\lambda \mathsf{x}.\,\mathsf{confine}_{\texttt{Unit}}\ (\mathsf{true}\ (\mathsf{protect}_{\texttt{Unit}}\ \mathsf{x}))))\ (\lambda \mathsf{y}.\,(\mathsf{y};\mathsf{unit})\ \mathsf{a}))) \hookrightarrow \\
(((\mathsf{f}\ (\lambda \mathsf{x}.\,\mathsf{confine}_{\texttt{Unit}}\ (\mathsf{true}\ (\mathsf{protect}_{\texttt{Unit}}\ \mathsf{x}))))\ (\mathsf{a};\mathsf{unit}))) \hookrightarrow \\
\text{if } \mathsf{a} \text{ is not } \mathsf{unit} \text{ then } \hookrightarrow \mathsf{wrong},\ \text{otherwise} \\
(((\mathsf{f}\ (\lambda \mathsf{x}.\,\mathsf{confine}_{\texttt{Unit}}\ (\mathsf{true}\ (\mathsf{protect}_{\texttt{Unit}}\ \mathsf{x}))))\ (\mathsf{unit}))) \hookrightarrow \\
(\mathsf{f}\ (\mathsf{confine}_{\texttt{Unit}}\ (\mathsf{true}\ (\mathsf{protect}_{\texttt{Unit}}\ \mathsf{unit})))) \hookrightarrow \\
(\mathsf{f}\ (\lambda \mathsf{y}.\,(\mathsf{y};\mathsf{unit})\ (\mathsf{true}\ (\lambda \mathsf{z}.\,\mathsf{z}\ \mathsf{unit})))) \hookrightarrow \\
(\mathsf{f}\ (\lambda \mathsf{y}.\,(\mathsf{y};\mathsf{unit})\ (\mathsf{true}\ \mathsf{unit}))) \hookrightarrow \\
(\mathsf{f}\ (\lambda \mathsf{y}.\,(\mathsf{y};\mathsf{unit})\ (\mathsf{wrong}))) \hookrightarrow \mathsf{wrong}
\end{array}$$

If $\mathbb{C}_{\mathsf{f}}$ were $[\cdot]\ \lambda \mathsf{x}.\,\mathsf{true}$, it would still be non-well-behaving and the reduction would reach the following point (this can be verified by replacing true with $\lambda \mathsf{x}.\,\mathsf{true}$ in the reduction above):

$$\begin{array}{l}
(\mathsf{f}\ (\lambda \mathsf{y}.\,(\mathsf{y};\mathsf{unit})\ (\lambda \mathsf{x}.\,\mathsf{true}\ \mathsf{unit}))) \hookrightarrow \\
(\mathsf{f}\ (\lambda \mathsf{y}.\,(\mathsf{y};\mathsf{unit})\ \mathsf{true})) \hookrightarrow \\
(\mathsf{f}\ ((\mathsf{true};\mathsf{unit}))) \hookrightarrow \\
(\mathsf{f}\ (\mathsf{wrong})) \hookrightarrow \mathsf{wrong}
\end{array}$$

$\boxdot$

Proof of Theorem 13 ($[\![\,\cdot\,]\!]^{\lambda^{\tau}}_{\lambda^{\mathsf{u}}}$ has fail-safe behaviour)

Proof. Intuition: all ill behaved interactions either reduce to wrong immediately or reduce to lambdas with some term that will reduce to wrong inside that, if unravelled, reduce to wrong.

In this case $\nexists \mathsf{t} \sim \mathsf{t}$ means: t is wrong or $\exists \mathbb{C}.\mathbb{C}\mathsf{t} \hookrightarrow {}^{*}\mathsf{wrong}$.

Induction on the structure of $\mathbb{C}$.

1. Base Case

   - $\mathbb{C} \equiv \cdot$ This cannot arise because $\cdot \in \mathit{WBctx}^{\lambda^{\tau}}_{\lambda^{\mathsf{u}}}$
   - $\mathbb{C} \equiv \mathsf{unit}$ This cannot arise because $\mathsf{unit} \in \mathit{WBctx}^{\lambda^{\tau}}_{\lambda^{\mathsf{u}}}$
   - $\mathbb{C} \equiv \mathsf{true}$ This cannot arise because $\mathsf{true} \in \mathit{WBctx}^{\lambda^{\tau}}_{\lambda^{\mathsf{u}}}$

- $\mathbb{C} \equiv \mathsf{false}$ This cannot arise because $\mathsf{false} \in \mathit{WBctx}^{\lambda^{\tau}}_{\lambda^{\mathsf{u}}}$
- $\mathbb{C} \equiv \mathsf{wrong}$ The case holds.

2. Inductive case The IH is: $\mathbb{C}' \notin \mathit{WBctx}^{\mathcal{S}}_{\mathcal{T}}, \forall \mathbf{C}$, if $\mathbb{C}'[\![\mathbf{C}]\!]^{\mathcal{S}}_{\mathcal{T}}$, then $\mathbb{C}'[\![\mathbf{C}]\!]^{\mathcal{S}}_{\mathcal{T}} \hookrightarrow {}^{*}\mathsf{t}$ and $\nexists \mathbf{t}.\mathbf{t} \sim \mathsf{t}$.

   The IH is always applicable because if $\mathbb{C}[\![\mathbf{C}]\!]^{\mathcal{S}}_{\mathcal{T}}$, then $\mathbb{C}'[\![\mathbf{C}]\!]^{\mathcal{S}}_{\mathcal{T}}$.

   - $\mathbb{C} \equiv \lambda\mathsf{x}.\,\mathbb{C}'$ This holds by IH.
   - $\mathbb{C} \equiv \mathsf{t}\ \mathbb{C}'$ Analysis on t:
     - $\mathsf{t} \hookrightarrow {}^{*}\mathsf{v}$ Analysis on v
       - $\mathsf{v} \equiv \mathsf{unit}, \mathsf{true}, \mathsf{false}, \langle \mathsf{v}, \mathsf{v} \rangle, \mathrm{inl}\ \mathsf{v}, \mathrm{inr}\ \mathsf{v}$, the term reduces to wrong, so this case holds.
       - $\mathsf{v} \equiv \lambda\mathsf{x}.\,\mathsf{t}'$ Carry a second general induction on t'.
         - $\mathsf{t}' \equiv \mathsf{unit}, \mathsf{true}, \mathsf{false}$: these cases cannot arise because they are not in $\mathit{WBctx}^{\lambda^{\tau}}_{\lambda^{\mathsf{u}}}$.
           For example, $\lambda\mathsf{x}.\,\mathsf{unit}\ \mathbb{C}' \sim \boldsymbol{\lambda}\mathbf{x}.\,\mathbf{unit}\ \mathbf{unit}$.
         - $\mathsf{t}' \equiv \mathsf{x}$ This holds by IH because $\lambda\mathsf{x}.\,\mathsf{x}\ \mathbb{C}' \hookrightarrow \mathbb{C}'$.
         - $\mathsf{t}' \equiv \mathsf{wrong}$ This holds trivially because $\lambda\mathsf{x}.\,\mathsf{wrong}\ \mathbb{C}' \hookrightarrow \mathsf{wrong}$.
         - All other inductive cases: They hold by the second IH.
     - $\mathsf{t} \not\hookrightarrow^{*} \mathsf{v}$ The term reduces to wrong, so this case holds.
     - $\mathsf{t}\Uparrow$ The term is related to $\mathbf{omega}\ \mathbf{unit}$ so this case does not arise.
   - $\mathbb{C} \equiv \mathbb{C}'\ \mathsf{t}$ This holds by IH.
   - $\mathbb{C} \equiv \mathbb{C}'.1$ This holds by IH.
   - $\mathbb{C} \equiv \mathbb{C}'.2$ This is analogous to the previous case.
   - $\mathbb{C} \equiv \mathrm{inl}\ \mathbb{C}'$ This holds by IH.
   - $\mathbb{C} \equiv \mathrm{inr}\ \mathbb{C}'$ This is analogous to the previous case.
   - $\mathbb{C} \equiv \langle \mathbb{C}', \mathsf{t} \rangle$ This holds by IH.
   - $\mathbb{C} \equiv \langle \mathsf{t}, \mathbb{C}' \rangle$ Analysis on t:
     - $\mathsf{t} \hookrightarrow {}^{*}\mathsf{v}$ This holds by IH.
     - $\mathsf{t} \not\hookrightarrow^{*} \mathsf{v}$ The term reduces to wrong, so this case holds.
     - $\mathsf{t}\Uparrow$ The term is related to $\langle \mathbf{omega}, \mathbf{unit} \rangle$ so this case does not arise.
   - $\mathbb{C} \equiv \mathsf{t};\mathbb{C}'$ Analysis on t:
     - $\mathsf{t} \hookrightarrow {}^{*}\mathsf{v}$ Analysis on v
       - $\mathsf{v} \equiv \lambda\mathsf{x}.\,\mathsf{t}', \mathsf{true}, \mathsf{false}, \langle \mathsf{v}, \mathsf{v} \rangle, \mathrm{inl}\ \mathsf{v}, \mathrm{inr}\ \mathsf{v}$, the term reduces to wrong, so this case holds.
       - unit This holds by IH.
     - $\mathsf{t} \not\hookrightarrow^{*} \mathsf{v}$ The term reduces to wrong, so this case holds.
     - $\mathsf{t}\Uparrow$ The term is related to $\mathbf{omega}; \mathbf{unit}$ so this case does not arise.

- $\mathbb{C} \equiv \mathbb{C}'; \mathsf{t}$ This holds by IH.
- $\mathbb{C} \equiv \text{case } \mathbb{C}' \text{ of inl } \mathsf{x}_1 \mapsto \mathsf{t}_1 \mid \text{inr } \mathsf{x}_2 \mapsto \mathsf{t}_2$ This holds by IH.
- $\mathbb{C} \equiv \text{case } \mathsf{t} \text{ of inl } \mathsf{x}_1 \mapsto \mathbb{C}' \mid \text{inr } \mathsf{x}_2 \mapsto \mathsf{t}_2$ Analysis on t:
  - $\mathsf{t} \hookrightarrow{}^{*} \mathsf{v}$ Analysis on v
    - $\mathsf{v} \equiv \lambda \mathsf{x}.\, \mathsf{t}'$, true, false, $\langle \mathsf{v}, \mathsf{v} \rangle$, unit, the term reduces to wrong, so this case holds.
    - $\text{inl } \mathsf{v}$ This holds by IH.
    - $\text{inr } \mathsf{v}$ Carry on another general induction in t_2.
      - unit The term is related to $\text{case inr } \mathbf{v} \text{ of inl } \mathbf{x_1} \mapsto \mathbf{unit} \mid \text{inr } \mathbf{x_2} \mapsto \mathbf{unit}$.
      - false The term is related to $\text{case inr } \mathbf{v} \text{ of inl } \mathbf{x_1} \mapsto \mathbf{unit} \mid \text{inr } \mathbf{x_2} \mapsto \mathbf{false}$.
      - true The term is related to $\text{case inr } \mathbf{v} \text{ of inl } \mathbf{x_1} \mapsto \mathbf{unit} \mid \text{inr } \mathbf{x_2} \mapsto \mathbf{true}$.
      - $\lambda \mathsf{x}.\, \mathsf{t}'$ This holds by IH on t'.
      - Inductive case: all cases hold by the second IH.
  - $\mathsf{t} \not\hookrightarrow{}^{*} \mathsf{v}$ The term reduces to wrong, so this case holds.
  - $\mathsf{t} \Uparrow$ The term is related to $\text{case } \mathbf{omega} \text{ of inl } \mathbf{x_1} \mapsto \mathbf{unit} \mid \text{inr } \mathbf{x_2} \mapsto \mathbf{unit}$ so this case does not arise.
- $\mathbb{C} \equiv \text{case } \mathsf{t} \text{ of inl } \mathsf{x}_1 \mapsto \mathsf{t}_1 \mid \text{inr } \mathsf{x}_2 \mapsto \mathbb{C}'$ This is analogous to the previous case.
- $\mathbb{C} \equiv \text{if } \mathbb{C}' \text{ then } \mathsf{t}_1 \text{ else } \mathsf{t}_2$ This holds by IH.
- $\mathbb{C} \equiv \text{if } \mathsf{t} \text{ then } \mathbb{C}' \text{ else } \mathsf{t}_2$ Analysis on t:
  - $\mathsf{t} \hookrightarrow{}^{*} \mathsf{v}$ Analysis on v
    - $\mathsf{v} \equiv \lambda \mathsf{x}.\, \mathsf{t}'$, $\text{inl } \mathsf{v}$, $\text{inr } \mathsf{v}$, $\langle \mathsf{v}, \mathsf{v} \rangle$, unit, the term reduces to wrong, so this case holds.
    - true This holds by IH.
    - false This case is analogous to the $\text{inr } \mathsf{v}$ case 3 steps before.
  - $\mathsf{t} \not\hookrightarrow{}^{*} \mathsf{v}$ The term reduces to wrong, so this case holds.
  - $\mathsf{t} \Uparrow$ The term is related to $\text{if } \mathbf{omega} \text{ then } \mathbf{unit} \text{ else } \mathbf{unit}$ so this case does not arise.
- $\mathbb{C} \equiv \text{if } \mathsf{t} \text{ then } \mathsf{t}_1 \text{ else } \mathbb{C}'$ This is analogous to the previous case. □

Proof of Theorem 14 ($\llbracket \cdot \rrbracket^{\lambda^{\tau}}_{\lambda^{\mathsf{u}}}$ is trace-preserving)

Proof. By Theorem 12 and Theorem 13 and Theorem 11. □

B Additional Material for $[\![\cdot]\!]^{\lambda^\tau}_{\lambda^{\mathsf{u}}}$

B.1 Syntax of the Languages of $[\![\cdot]\!]^{\lambda^\tau}_{\lambda^{\mathsf{u}}}$

$$
\begin{aligned}
\mathbf{t} ::= {} & \mathbf{unit} \mid \mathbf{true} \mid \mathbf{false} \mid \lambda\mathbf{x} : \tau.\ \mathbf{t} \mid \mathbf{x} \mid \mathbf{t}\ \mathbf{t} \mid \mathbf{t}.\mathbf{1} \mid \mathbf{t}.\mathbf{2} \mid \langle \mathbf{t}, \mathbf{t} \rangle \\
& \mid\ \mathrm{inl}\ \mathbf{t} \mid \mathrm{inr}\ \mathbf{t} \mid \mathrm{case}\ \mathbf{t}\ \mathrm{of}\ \mathrm{inl}\ \mathbf{x_1} \mapsto \mathbf{t} \mid \mathrm{inr}\ \mathbf{x_2} \mapsto \mathbf{t} \mid \mathbf{t}; \mathbf{t} \\
& \mid\ \mathrm{if}\ \mathbf{t}\ \mathrm{then}\ \mathbf{t}\ \mathrm{else}\ \mathbf{t} \mid \mathrm{fix}_{\tau \to \tau}\ \mathbf{t} \\
\mathbf{v} ::= {} & \mathbf{unit} \mid \mathbf{true} \mid \mathbf{false} \mid \lambda\mathbf{x} : \tau.\ \mathbf{t} \mid \langle \mathbf{v}, \mathbf{v} \rangle \mid \mathrm{inl}\ \mathbf{v} \mid \mathrm{inr}\ \mathbf{v} \\
\tau ::= {} & \mathtt{Unit} \mid \mathtt{Bool} \mid \tau \to \tau \mid \tau \times \tau \mid \tau \uplus \tau
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{t} ::= {} & \mathsf{unit} \mid \mathsf{true} \mid \mathsf{false} \mid \lambda\mathsf{x}.\,\mathsf{t} \mid \mathsf{x} \mid \mathsf{t}\ \mathsf{t} \mid \mathsf{t.1} \mid \mathsf{t.2} \mid \langle \mathsf{t}, \mathsf{t} \rangle \\
& \mid\ \mathrm{inl}\ \mathsf{t} \mid \mathrm{inr}\ \mathsf{t} \mid \mathrm{case}\ \mathsf{t}\ \mathrm{of}\ \mathrm{inl}\ \mathsf{x}_1 \mapsto \mathsf{t} \mid \mathrm{inr}\ \mathsf{x}_2 \mapsto \mathsf{t} \mid \mathsf{t}; \mathsf{t} \\
& \mid\ \mathrm{if}\ \mathsf{t}\ \mathrm{then}\ \mathsf{t}\ \mathrm{else}\ \mathsf{t} \mid \mathsf{wrong} \\
\mathsf{v} ::= {} & \mathsf{unit} \mid \mathsf{true} \mid \mathsf{false} \mid \lambda\mathsf{x}.\,\mathsf{t} \mid \langle \mathsf{v}, \mathsf{v} \rangle \mid \mathrm{inl}\ \mathsf{v} \mid \mathrm{inr}\ \mathsf{v}
\end{aligned}
$$

B.2 The $\mathtt{protect}(\cdot)$ Function

See Figure 3.

$$
\mathsf{protect}_{\mathtt{Unit}} \overset{\mathsf{def}}{=} \lambda\mathsf{x}.\,\mathsf{x} \qquad\qquad \mathsf{protect}_{\mathtt{Bool}} \overset{\mathsf{def}}{=} \lambda\mathsf{x}.\,\mathsf{x}
$$

$$
\begin{aligned}
\mathsf{protect}_{\tau_1 \times \tau_2} &\overset{\mathsf{def}}{=} \lambda\mathsf{y}.\,\langle \mathsf{protect}_{\tau_1}\ \mathsf{y.1}, \mathsf{protect}_{\tau_2}\ \mathsf{y.2} \rangle \\
\mathsf{protect}_{\tau_1 \uplus \tau_2} &\overset{\mathsf{def}}{=} \lambda\mathsf{y}.\,\mathrm{case}\ \mathsf{y}\ \mathrm{of} \left| \begin{array}{l} \mathrm{inl}\ \mathsf{x} \mapsto \mathrm{inl}\ (\mathsf{protect}_{\tau_1}\ \mathsf{x}) \\ \mathrm{inr}\ \mathsf{x} \mapsto \mathrm{inr}\ (\mathsf{protect}_{\tau_2}\ \mathsf{x}) \end{array} \right. \\
\mathsf{protect}_{\tau_1 \to \tau_2} &\overset{\mathsf{def}}{=} \lambda\mathsf{y}.\,\lambda\mathsf{x}.\mathsf{protect}_{\tau_2}\ (\mathsf{y}\ (\mathsf{confine}_{\tau_1}\ \mathsf{x})) \\
\mathsf{confine}_{\mathtt{Unit}} &\overset{\mathsf{def}}{=} \lambda\mathsf{y}.\,(\mathsf{y}; \mathsf{unit}) \\
\mathsf{confine}_{\mathtt{Bool}} &\overset{\mathsf{def}}{=} \lambda\mathsf{y}.\,\mathrm{if}\ \mathsf{y}\ \mathrm{then}\ \mathsf{true}\ \mathrm{else}\ \mathsf{false} \\
\mathsf{confine}_{\tau_1 \times \tau_2} &\overset{\mathsf{def}}{=} \lambda\mathsf{y}.\,\langle \mathsf{confine}_{\tau_1}\ \mathsf{y.1}, \mathsf{confine}_{\tau_2}\ \mathsf{y.2} \rangle \\
\mathsf{confine}_{\tau_1 \uplus \tau_2} &\overset{\mathsf{def}}{=} \lambda\mathsf{y}.\,\mathrm{case}\ \mathsf{y}\ \mathrm{of} \left| \begin{array}{l} \mathrm{inl}\ \mathsf{x} \mapsto \mathrm{inl}\ (\mathsf{confine}_{\tau_1}\ \mathsf{x}) \\ \mathrm{inr}\ \mathsf{x} \mapsto \mathrm{inr}\ (\mathsf{confine}_{\tau_2}\ \mathsf{x}) \end{array} \right. \\
\mathsf{confine}_{\tau_1 \to \tau_2} &\overset{\mathsf{def}}{=} \lambda\mathsf{y}.\,\lambda\mathsf{x}.\,\mathsf{confine}_{\tau_2}\ (\mathsf{y}\ (\mathsf{protect}_{\tau_1}\ \mathsf{x}))
\end{aligned}
$$

Figure 3: The $\mathtt{protect}()$ function for $[\![\cdot]\!]^{\lambda^\tau}_{\lambda^{\mathsf{u}}}$.

C TPC and Modular Compilation

C.1 Modular TPC

Inspired by the recent developments in secure compilation for modular compilers [42, 30], this section presents *Modular* **TPC** or *MTP*. Then it compares *MTP* with the modular version of compiler full abstraction (Appendix C.2).

First, we clarify what we mean by modular compiler. A modular compiler is one that applies itself recursively to the sub-components of a component and then links the compiled results to obtain a compiled component. This approach to compilation is standard. When we have a program that relies on libraries and something changes in the program, we do not recompile the libraries. Instead, we recompile just the program and re-link the result against the already compiled libraries.

Intuitively, a trace-preserving compiler must be secure when applied to programs that are composed of sets of components and not just a single component. So, assume that a component is made of sub-components. To make this explicit in the syntax, we write M for a sequence of linked components: $M = C_1 + C_2 + \cdots + C_n$, where $+$ is linking. We define *MTP* as follows.

Definition 25 (Modular **TPC**). $[\![\cdot]\!]^{\mathcal{S}}_{\mathcal{T}} \in MTP \overset{\mathsf{def}}{=} \forall \mathbf{M}, \mathbf{M}',$

$$\mathsf{TR}([\![\mathbf{M}]\!]^{\mathcal{S}}_{\mathcal{T}} + [\![\mathbf{M}']\!]^{\mathcal{S}}_{\mathcal{T}}) = \mathsf{TR}([\![\mathbf{M}+\mathbf{M}']\!]^{\mathcal{S}}_{\mathcal{T}}) \text{ and } [\![\cdot]\!]^{\mathcal{S}}_{\mathcal{T}} \in TP$$

This definition requires the compiler to be *TP* as well as modular, i.e., linking in the source and then compiling is equivalent to compiling and then linking in the target. The intuition behind the definition is straightforward: When compiling the two modules as a single unit $\mathbf{M}+\mathbf{M}'$, it prevents the compiler from generating code that causes an interaction between the two components that would not have existed had $\mathbf{M}$ and $\mathbf{M}$ been compiled separately.

Example 10 explains why it is not sufficient for just the compiler to be *TP*.

Example 10 (Need for Modular **TPC**). Consider a component $\mathbf{M}$ made of two sub-parts: $\mathbf{C_{counter}}$ and $\mathbf{C_{log}}$ so $\mathbf{M} \overset{\mathsf{def}}{=} \mathbf{C_{counter}} + \mathbf{C_{log}}$.

Intuitively, $\mathbf{C_{counter}}$ stores a counter for every other component in the system. $\mathbf{C_{counter}}$ can be called by another component to increment the its own counter (assume no other component can increment another component's counter) and any component can ask the status of the counter of another component. $\mathbf{C_{log}}$ logs accesses to resources. Any time the log is called, it calls for an increment in the counter, receives the increment, and then returns to whoever called it in the first place.

Traces of $\mathbf{C_{counter}}$ include these traces:

$$\begin{aligned}
\boldsymbol{\alpha}_{\mathbf{inc}} &= \ \mathbf{inc(Log)?} \cdot \mathbf{ret(Log,1)!} \cdot \ \mathbf{inc(Log)?} \cdot \mathbf{ret(Log,2)!} \cdots \\
\boldsymbol{\alpha}_{\mathbf{get}} &= \ \mathbf{get(Log)?} \cdot \mathbf{ret(0)!} \cdot \ \mathbf{get(Log)?} \cdot \mathbf{ret(1)!} \cdots
\end{aligned}$$

Traces of $\mathbf{C_{log}}$ include this trace:

$$\boldsymbol{\alpha}_{\mathbf{log}} = \ \mathbf{log(n)?} \cdot \mathbf{call\ inc(Log)!} \cdot \mathbf{ret(Log,1)?} \cdot \mathbf{ret(unit)!} \cdots$$

When cosidering $\mathbf{M}$ as a whole, however, interactions between $\mathbf{C_{log}}$ and $\mathbf{C_{counter}}$ are hidden. The traces of $\mathbf{M}$ are obtained by joining those of $\mathbf{C_{counter}}$ and $\mathbf{C_{log}}$. Specifically, they no longer have $\boldsymbol{\alpha}_{\mathbf{log}}$ above, but they include $\boldsymbol{\alpha}'_{\mathbf{log}}$ below:

$$\boldsymbol{\alpha}'_{\mathbf{log}} = \ \mathbf{log(n)?} \cdot \mathbf{ret(unit)!} \cdots$$

A compiler could compile sec M so that $\mathbf{C_{log}}$ calls to $\mathbf{C_{counter}}$ even on invalid actions. This would be problematic as the counter for $\mathbf{C_{log}}$ would be increased for invalid interactions as well (one can imagine this is used for some billing system where bills can thus be tampered with, making one pay for unreceived services). If we just reason about $\mathbf{C_{log}}$ or $\mathbf{C_{counter}}$ individually, this compilation is still TP. However, we want to prevent these compilers from being categorised as secure.

When we force the compiler to be modular and we reason about $\mathbf{C_{log}}+\mathbf{C_{counter}}$, that compiler is no longer TP. Thus the addition of the modularity requirement in MTP.

Without the modularity requirement, the following would be a valid target-level trace, where (.) identifies when the log would call and increment the counter:

$$\alpha_{\mathsf{trick}} = \ \mathsf{log(true)?} \cdot (.) \surd \ \mathsf{get(Log)?} \cdot \mathsf{ret(1)!} \cdots$$

However, for this to be a "valid" invalid trace, it must be a trace that relates to a source-level one interleaved with a possible combination of "invalid action - $\surd$". Because in the semantics of $\mathbf{M}$ there is no action of the form $\mathbf{get(Log)?} \cdot \mathbf{ret(1)!} \cdots$ (the counter starts as 0), trace α_{trick} is not a "valid" invalid trace that a MTP compiler allows.

With MTP, this problem does not arise, so the counter is only incremented on valid interactions and we can conclude that a MTP compiler preserves invariants shared between components. ⊡

C.2 Modularity for *FA* and TPC

Recent developments in secure compilation highlight that for modular compilers from a typed to an untyped language, full abstraction is not enough [42, 30]. If a fully abstract compiler is used to compile sub-components individually and then link them together, the resulting (whole) component can be subject to security exploits such as non well-bracketed flow of execution. For a modular compiler to be secure, it must be proven to be modular-fully abstract (Definition 26). Let MFA be the set of compilers that are modular fully abstract.

Definition 26 (Modular full abstraction). $[\![\cdot]\!]^{\mathcal{S}}_{\mathcal{T}} \in MFA \overset{\mathsf{def}}{=} \forall \mathbf{M_1}, \mathbf{M_2}, \mathbf{M_3}, \mathbf{M_4}.$ $\mathbf{M_1}+\mathbf{M_2} \simeq_{ctx} \mathbf{M_3}+\mathbf{M_4} \iff [\![\mathbf{M_1}]\!]^{\mathcal{S}}_{\mathcal{T}}+[\![\mathbf{M_2}]\!]^{\mathcal{S}}_{\mathcal{T}} \simeq_{ctx} [\![\mathbf{M_3}]\!]^{\mathcal{S}}_{\mathcal{T}}+[\![\mathbf{M_4}]\!]^{\mathcal{S}}_{\mathcal{T}}.$

Modular trace-preserving compilation and modular full abstraction are in the same relationship that (non-modular) trace-preserving compilation and full abstraction are. *MTP* in fact implies *MFA* by much of the same argument that **TPC** implies **FAC**. *MFA*, on the other side, does not imply *MTP* but, with the addition of *FSB* implies the halting variant of *MTP*.

Theorem 17 (Relationship between *MFA* and *MTP*)**.** $\forall \llbracket \cdot \rrbracket^{\mathcal{S}}_{\mathcal{T}}\ \llbracket \cdot \rrbracket^{\mathcal{S}}_{\mathcal{T}} \in \mathit{MTP} \Rightarrow \llbracket \cdot \rrbracket^{\mathcal{S}}_{\mathcal{T}} \in \mathit{MFA}$ and $\llbracket \cdot \rrbracket^{\mathcal{S}}_{\mathcal{T}} \in \mathit{MFA} \not\Rightarrow \llbracket \cdot \rrbracket^{\mathcal{S}}_{\mathcal{T}} \in \mathit{MTP}$.