Fully Abstract and Robust Compilation

And How to Reconcile the Two, Abstractly

Carmine Abate[1], Matteo Busi[2], and Stelios Tsampas[3](✉)

[1] MPI-SP Bochum, Bochum, Germany
carmine.abate@mpi-sp.org
[2] Università di Pisa, Pisa, Italy
matteo.busi@di.unipi.it
[3] KU Leuven, Leuven, Belgium
stelios.tsampas@cs.kuleuven.be

Abstract. The most prominent formal criterion for secure compilation is *full abstraction*, the preservation and reflection of contextual equivalence. Recent work introduced *robust compilation*, defined as the preservation of *robust satisfaction* of hyperproperties, i.e., their satisfaction against arbitrary attackers. In this paper, we initially set out to compare these two approaches to secure compilation. To that end, we provide an exact description of the hyperproperties that are robustly satisfied by programs compiled with a fully abstract compiler, and show that they can be meaningless or trivial. We then propose a novel criterion for secure compilation formulated in the framework of Mathematical Operational Semantics (MOS), guaranteeing both full abstraction and the preservation of robust satisfaction of hyperproperties in a more sensible manner.

Keywords: Secure compilation · Fully abstract compilation · Robust hyperproperty preservation · Language-based security · Mathematical Operational Semantics

Remark. To ease reading, we highlight the elements of source languages in blue, sans-serif, the target elements in **red, bold** and the common ones in black [33].

1 Introduction

Due to the complexity of modern computing systems, engineers make substantial use of *layered design*. Higher layers hide details of the lower ones and come with abstractions that ease reasoning about the system itself [39]. A layered design of programming languages allows to benefit from modules, interfaces or dependent types of a *source, high-level* language to write well-structured programs, and execute them efficiently in a *target, low-level* language, after *compilation*. Unfortunately, an attacker may exploit the lack of abstractions at the low-level to mount a so-called *layer-below attack* [39], which is otherwise impossible at the high-level [17,18].

H. Oh (Ed.): APLAS 2021, LNCS 13008, pp. 83–101, 2021.
https://doi.org/10.1007/978-3-030-89051-3_6

Secure compilation [35] devises both principles and proof techniques to preserve the (security-relevant) abstractions of the source and prevent layer-below attacks. Abadi [1] hinted that secure compilers must respect *equivalences*, as some security properties can be expressed in terms of indistinguishability w.r.t. arbitrary attackers, or *contextual equivalence*. Fully abstract compilers preserve and reflect (to avoid trivial translations) contextual equivalence.

Two decades of successes [1,8,9,13,14,19,34,36,43,45] made full abstraction the gold-standard for secure compilation. However, some ad-hoc examples from recent literature [4,37] showed that fully abstract compilers may still introduce bugs that were not present in source programs, e.g.,

Example 1 (See also Appendix E.5 of [4]). Consider source programs to be functions $\mathbb{B} \to \mathbb{N}$ (from booleans to natural numbers) and target ones to be functions $\mathbb{N} \to \mathbb{N}$. Define contextual equivalence to be equality of outputs on equal inputs. Next, identify $\mathbb{B}$ with $\{0, 1\} \subseteq \mathbb{N}$, and compile a program P to $[\![\mathsf{P}]\!] : \mathbb{N} \to \mathbb{N}$ that coincides with $\mathsf{P} : \mathbb{B} \to \mathbb{N}$ on $\{0, 1\}$ and returns a default value – denoting a bug – otherwise,

$$[\![\mathsf{P}]\!](n) = \begin{cases} \mathsf{P}(n) & \text{for } n = 0, 1 \\ 42 & \text{otherwise} \end{cases}$$

$[\![\cdot]\!]$ is fully abstract, yet a source program that "never outputs 42", will no longer enjoy this property. ■

This simple example underlines the fact that if a security property like "never output 42" is not captured by contextual equivalence, there is no guarantee it will be preserved by a fully abstract compiler. Abadi [1] tellingly wrote

> [. . .] we still have only a limited understanding of how to specify and prove that a translation preserves particular security properties. [. . .]

Abate et al. [4] proposed to specify security in terms of *hyperproperties*, sets of sets of traces of observable events [15]. In this setting, they consider a compiler *secure* only if it *robustly preserves* a class of hyperproperties, i.e., if it preserves their satisfaction against arbitrary attackers. For Example 1, "never output 42" can be specified as a *safety* hyperproperty, where function inputs and outputs are the observable events. The above compiler $[\![\cdot]\!]$ is *not* secure according to Abate et al. [4], as it does not robustly preserve the class of safety hyperproperties. More generally, each particular class of hyperproperties, e.g., the one for data integrity or the one for data confidentiality [15], determines a precise formal secure compilation criterion.

Despite the introduction of the robust criteria, full abstraction is still widely adopted [14,19,43,45], for at least two reasons. First, contextual equivalence can model security properties such as noninterference [13], isolation [14], well-bracketed control flow or local state encapsulation [43] for programs that don't expose events externally. Second, even though fully abstract compilers do not *in general* preserve data integrity or confidentiality, they often do so in practice.

Fully abstract and robust compilation both embody valuable notions of secure compilation and neither is stronger than the other nor are they orthogonal, which makes us believe their relation deserves further investigation. Our goal is to have criteria with well understood security guarantees for compiled programs, so that both users and developers of compilers may decide which criterion better fits their needs. For that, we assume an abstract trace semantics, collecting observables events and internal steps, is given for both source and target languages, and start our quest not by asking *if* a given fully abstract compiler preserves *all* hyperproperties, but *which ones do* and *which ones do not* preserve.

Contributions. First, we make explicit the guarantees given by full abstraction w.r.t. arbitrary source hyperproperties. We achieve this by showing that for every fully abstract compiler $[\![\cdot]\!]$, there exists a translation or interpretation of source hyperproperties into target ones, $\tilde{\tau}$, such that if P robustly satisfies a source hyperproperty H, $[\![\mathsf{P}]\!]$ robustly satisfies $\tilde{\tau}(\mathsf{H})$ (Theorem 1). However, we observe that a fully abstract compiler may fail to preserve the robust satisfaction of some hyperproperty, as $\tilde{\tau}$ may map interesting hyperproperties to trivial ones (Example 2). We then provide a sufficient and necessary condition to preserve the robust satisfaction of hyperproperties (Corollary 1), but argue that it is unfeasible to be proven true for an arbitrary fully abstract compiler. To overcome the above issues, we introduce a novel criterion, that we formulate in the abstract framework of Mathematical Operational Semantics (MOS). We show that our novel criterion implies full abstraction and the preservation of robust satisfaction of arbitrary hyperproperties (Sect. 5). We illustrate effectiveness and realizability of our criterion in Example 3.

2 Fully Abstract and Robust Compilation

Let us briefly recall the intuition of fully abstract and robust compilation, and provide their rigorous definitions. We refer the interested reader to [3,4,35] for more details.

2.1 Fully Abstract Compilation

Abadi [1] proposed fully abstract compilation to preserve security properties such as confidentiality and integrity when these are expressed in terms of indistinguishability w.r.t. the observations of arbitrary attackers, the latter modeled as execution contexts. For a concrete example, if no source context $\mathsf{C_S}$ can distinguish a program P_1 that uses some confidential data k from a program P_2 that does not, we can deduce that k is kept confidential by P_1. Thus, a compiler $[\![\cdot]\!]$ from a source language to a target one, that aims to preserve confidentiality, must ensure that also $[\![\mathsf{P}_1]\!]$ and $[\![\mathsf{P}_2]\!]$ are equivalent w.r.t. the observations of any target context $\mathbf{C_T}$. To avoid trivial translations, one typically asks for the reflection of the equivalence as well.

Definition 1 (Fully abstract compilation [1]). *A compiler* $[\![\cdot]\!]$ *is* fully abstract *iff for any* P_1 *and* P_2*,*

$$(\forall \mathsf{C_S}.\mathsf{C_S}[\mathsf{P_1}] \approx \mathsf{C_S}[\mathsf{P_2}]) \Leftrightarrow (\forall \mathbf{C_T}.\mathbf{C_T}[[\![\mathsf{P_1}]\!]] \approx \mathbf{C_T}[[\![\mathsf{P_2}]\!]])$$

where $\mathsf{C_S}$, $\mathbf{C_T}$ *denote source and target contexts resp.,* $\approx$, $\approx$ *denote the two contextual equivalences, i.e., equivalence relations on programs.*

Notice that the security notions one can preserve and reflect with a fully abstract compiler are those captured by the contextual equivalence relation $\approx$, that determines both the meaningfulness and the effectiveness of full abstraction. Indeed, if $\approx$ is too coarse-grained, some interesting security properties may be ignored. Dually, if $\approx$ is too fine-grained, equivalent source programs may not have counterparts that are equivalent in the target. In Sect. 3, we pick $\approx$ to be equality of execution traces which, under mild assumptions [20,28], coincides with other common choices of $\approx$ (see also Sect. 6).

2.2 Robust Compilation

Abate et al. [4] suggest a family of secure compilation criteria that depend on the security notion one is interested in preserving. The key idea in their criteria is the preservation of *robust satisfaction*, i.e., satisfaction of (classes of) security properties against arbitrary attackers, modeled as contexts. More concretely, Abate et al. [3,4] assume that every execution of a program exposes a trace of observable events $t \in \mathit{Trace}$ for a fixed set Trace and model interesting security notions like data integrity, confidentiality or observational determinism as sets of sets of traces, i.e., *hyperproperties* denoted by $H \in \wp(\wp(\mathit{Trace}))$ [15].

Definition 2 (Robust satisfaction [3,4]**).** *A program* P *robustly satisfies a hyperproperty* H *iff* $\forall C.\ C[P] \models H$*, where* $C[P] \models H \triangleq \mathit{beh}(C[P]) \in H$ *and* $\mathit{beh}(C[P])$ *is the set of all traces that can be observed when executing* $C[P]$*.*

Secure compilation criteria can then be defined as the preservation of robust satisfaction of classes of hyperproperties such as safety or liveness [4], in this paper we consider the class of *all* hyperproperties and *robust hyperproperty preservation* (RHP^τ from [3]). For that, consider a function τ that takes a source-level hyperproperty and returns its interpretation (or translation) at the target level. Intuitively, a compiler $[\![\cdot]\!]$ is RHP^τ if, for any source hyperproperty H robustly satisfied by P, its interpretation $\tau(\mathsf{H})$ is robustly satisfied by $[\![\mathsf{P}]\!]$, formally:

Definition 3 (Robust hyperproperty preservation). *A compiler* $[\![\cdot]\!]$ preserves the robust satisfaction of hyperproperties *according to a translation* $\tau : \wp(\wp(\mathsf{Trace_S})) \to \wp(\wp(\mathbf{Trace_T}))$ iff *the following* RHP^τ *holds*

$$\mathsf{RHP}^\tau \equiv \forall \mathsf{P}\ \forall \mathsf{H} \in \wp(\wp(\mathit{Trace})).\ (\forall \mathsf{C_S}.\ \mathsf{C_S}[\mathsf{P}] \models \mathsf{H}) \Rightarrow$$
$$(\forall \mathbf{C_T}.\ \mathbf{C_T}[[\![\mathsf{P}]\!]] \models \tau(\mathsf{H}))$$

when τ *is clear from the context we simply say that* $[\![\cdot]\!]$ *is robust.*

RHP^τ can be formulated without quantification on hyperproperties [3,4].

Lemma 1 (Property-free RHP^τ). *For a compiler $[\![\cdot]\!]$, RHP^τ is equivalent to*[1]

$$\forall \mathsf{P}\ \forall \mathbf{C_T}\ \exists \mathsf{C_S}.\ \mathbf{beh_T}(\mathbf{C_T}[[\![\mathsf{P}]\!]]) = \tau(\mathsf{beh_S}(\mathsf{C_S}[\mathsf{P}]))$$

Notice that, while Definition 3 describes—through τ—the target *guarantees* for $[\![\mathsf{P}]\!]$ against arbitrary target contexts, Lemma 1 enables proofs by *back-translation*. In fact, similarly to fully abstract compilation [35], one can prove that a compiler is RHP^τ by exhibiting a so-called *back-translation* map producing a source context $\mathsf{C_S}$ whose interaction with P exposes "the same" observables as $\mathbf{C_T}$ does with $[\![\mathsf{P}]\!]$:

Remark 1 (RHP^τ by back-translation). RHP^τ holds if there exists a back-translation function bk such that for any $\mathbf{C_T}$ and any P, $bk(\mathbf{C_T}[[\![\mathsf{P}]\!]]) = \mathsf{C_S}$ is such that $\mathbf{beh_T}(\mathbf{C_T}[[\![\mathsf{P}]\!]]) = \tau(\mathsf{beh_S}(\mathsf{C_S}[\mathsf{P}]))$.

3 Comparing FAC and RHP^τ

In the previous section we defined fully abstract compilation as the preservation and reflection of contextual equivalence, i.e., what the contexts can observe about programs. Instead, RHP^τ was defined as the preservation of (robust satisfaction of) hyperproperties of externally observable traces of events. To enable any comparison, we first provide an intuition on how to accommodate the mismatch in *observations* between full abstraction and RHP^τ (see the online appendix [5] for all the details). We assume the operational semantics of our languages exhaustively specify the execution of programs in contexts, including both internal steps and steps that expose externally observable *events* like inputs and outputs. Also, we say that a trace is *abstract* if it collects both internal steps and externally observable events. In a slight abuse of notation, we still denote with $beh(C[P])$ the set of all the possible *abstract* traces allowed by the semantics when executing P in C. Moreover, since hyperproperties just express predicates over events , we now write $beh(C[P]) \in H$ to mean that the traces of events for $C[P]$ satisfy the hyperproperty H. Finally, we elect to express contextual equivalence as the equality of the (sets of) abstract traces in an arbitrary context.

Definition 4 (Equality of $beh(\cdot)$). *For programs P_1, P_2 and a context C,*

$$C[P_1] \approx C[P_2] \iff beh(C[P_1]) = beh(C[P_2])$$

In Sect. 6 we discuss other common choices for $\approx$ such as *equi-termination*, and the hypotheses under which they are equivalent to ours. We now instantiate Definition 1 on the contextual equivalence from Definition 4 and make explicit the notion of fully abstract compilation we are going to use from now on. Note how we are only interested in the *preservation* of contextual equivalence, as reflection is often subsumed by compiler correctness (e.g., in absence of internal non-determinism) [1,35].

1 $\tau(\mathsf{beh_S}(\mathsf{C_S}[\mathsf{P}]))$ is a shorthand for $\tau(\{\mathsf{beh_S}(\mathsf{C_S}[\mathsf{P}])\})$.

Definition 5 (FAC). *For a compiler* $[\![\cdot]\!]$, FAC *is the following predicate*

$$\begin{aligned}\mathsf{FAC} \equiv \forall \mathsf{P}_1 \mathsf{P}_2 . (\forall \mathsf{C_S} .\ \mathsf{behs}(\mathsf{C_S}[\mathsf{P}_1]) = \mathsf{behs}(\mathsf{C_S}[\mathsf{P}_2])) \Rightarrow \\ (\forall \mathbf{C_T} .\ \mathbf{beh_T}(\mathbf{C_T}[[\![\mathsf{P}_1]\!]]) = \mathbf{beh_T}(\mathbf{C_T}[[\![\mathsf{P}_2]\!]]))\end{aligned}$$

Abate et al. [4], Patrignani and Garg [37] have previously investigated the relation between FAC as in Definition 5 and RHP$^\tau$. In particular, Abate et al. [4] showed that FAC does not imply any of the robust criteria, with an example similar to the one we sketched in Sect. 1. In this section, we provide further evidence of this fact: a fully abstract compiler that does not preserve the robust satisfaction of a security-relevant hyperproperty, namely noninterference. More details on the example can be found in the online appendix [5].

Example 2. Let Source and **Target** to be two WHILE-like languages [31] with a mutable state. A state $s \in S \triangleq (\mathit{Var} \to \mathbb{N})$ assigns every variable $\mathtt{v} \in \mathit{Var}$ a natural number. We assume Var to be partitioned into a "high" (private) and a "low" (public) part. We write $\mathtt{v} \in \mathit{Var}_H$ ($\mathtt{v} \in \mathit{Var}_L$, resp.) to denote that the variable $\mathtt{v}$ is private (public, resp.). Partial programs are defined in the same way in both Source and **Target**, whereas whole programs, or terms, are obtained by filling the hole(s) of a context with a partial program (Fig. 1). The only context in Source is $[\cdot]$, called the *identity* context and such that for any P, $[\mathsf{P}] = \mathsf{P}$. Instead, contexts in **Target** additionally include $\lceil\cdot\rceil$ that is able to observe the *internal* event $\mathcal{H}$ (intuitively, a form of *information leakage* that is not observed by source contexts) and report it by emitting **!**.

$$\begin{aligned}&\langle P\rangle ::= \text{ skip} \mid \mathtt{v} := \langle \mathit{expr}\rangle \mid \langle P\rangle ; \langle P\rangle \mid \text{while } \langle \mathit{expr}\rangle\ \langle P\rangle \\ &\langle \mathsf{C_S}\rangle ::= [\cdot] \qquad\qquad\qquad \langle \mathbf{C_T}\rangle ::= [\cdot] \mid \lceil\cdot\rceil\end{aligned}$$

Fig. 1. $\langle P\rangle$ defines the syntax of both Source and **Target** partial programs, where $\langle \mathit{expr}\rangle$ denotes the usual arithmetic expressions over $\mathbb{N}$. $\langle \mathsf{C_S}\rangle$ and $\langle \mathbf{C_T}\rangle$ define instead the contexts of Source and **Target**, respectively.

The semantics of Source and **Target** are partially given in Fig. 2. Rule asnL is for assignments that do not involve high variables. asnH is for assignments of high variables, and – upon a change in their value – the internal trace $\mathcal{H}$ is emitted. The **Target** counterparts, **asnL** and **asnH**, work similarly. Finally, the most interesting rule is **bang2**, where we see how context $\lceil\cdot\rceil$ reports a **!** upon encountering an $\mathcal{H}$.

$$\mathsf{asnL}\ \frac{\mathtt{v} \in \mathit{Var}_L \qquad e \cap \mathit{Var}_H = \emptyset}{s, \mathtt{v} := e \to s_{[\mathtt{v} \leftarrow [e]_s]}, \checkmark} \qquad \mathsf{asnH}\ \frac{\mathtt{v} \in \mathit{Var}_H \qquad s(\mathtt{v}) \neq [e]_s}{s, \mathtt{v} := e \xrightarrow{\mathcal{H}} s_{[\mathtt{v} \leftarrow [e]_s]}, \checkmark}$$

$$\mathbf{bang2}\ \frac{s, \mathbf{p} \xrightarrow{\mathcal{H}} s', \mathbf{p}'}{s, \lceil\ \mathbf{p}\ \rceil \xrightarrow{!} s', \mathbf{p}'}$$

Fig. 2. Selected rules of Source and **Target**.

For example, consider a high variable $\mathtt{v} \in \mathit{Var}_H$ and the Source program $\mathsf{P} \triangleq \mathtt{v} := 42$. When P is plugged in the identity context $[\cdot]$, the resulting behavior is $\mathsf{behs_S}([\mathsf{P}]) = \left\{s \cdot \mathcal{H} \cdot s' \cdot \checkmark \;\middle|\; s \in S \wedge s' = s_{[\mathtt{v} \leftarrow 42]}\right\}$. Intuitively, the traces in $\mathsf{behs_S}([\mathsf{P}])$ express that the execution starts in a state s, then a high variable is updated ($\mathcal{H}$) leading to state s' and then the program terminates ($\checkmark$). For the same $\mathtt{v} \in \mathit{Var}_H$, target program $\mathbf{P} \triangleq \mathtt{v} := 42$ in $\lceil \cdot \rceil$, we have that $\mathbf{beh_T}(\lceil \mathbf{P} \rceil) = \left\{s \cdot ! \cdot s' \cdot \checkmark \;\middle|\; s \in S \wedge s' = s_{[\mathtt{v} \leftarrow 42]}\right\}$. Notice the additional $!$ w.r.t. the source, due to the fact that the context observed a change in a high variable. Informally, we say that a program satisfies noninterference if, executing it in two low-equivalent initial states, it transitions to two low-equivalent states. More rigorously, noninterference can be defined for both Source and **Target** as the following hyperproperty $\mathit{NI} \in \wp(\wp(\mathit{Trace}))$,

$$\mathit{NI} = \left\{\pi \in \wp(\mathit{Trace}) \;\middle|\; \forall t_1, t_2 \in \pi.\; t_1^0 =_L t_2^0 \Rightarrow t_1 =_L t_2\right\}$$

where t_i^0 stands for the first observable of the trace t_i and $=_L$ denotes the fact that two states are low-equivalent (i.e., they coincide on all $x \in \mathit{Var}_L$). Also, we lift the notation to traces and write $t_1 =_L t_2$ to denote that t_1 and t_2 are pointwise low-equivalent. More precisely, $=_L$ ignores all occurrences of $\mathcal{H}$ (as it is internal) and compares traces observable-by-observable, relating $\checkmark$ and $!$ to themselves and comparing states with the above notion of low-equivalence.

The identity compiler preserves trace equality (see the online appendix [5] for the proof), but does not preserve the robust satisfaction of noninterference as the **Target** context $\lceil \cdot \rceil$ can detect changes in high variables and report a $!$. ■

On the one hand, RHP^{τ} provides an explicit description of the target hyperproperty $\tau(\mathsf{H})$ that is guaranteed to be robustly satisfied after compilation under the hypothesis that H is robustly satisfied in the source. However, RHP^{τ} does not imply the preservation of contextual equivalence (or trace equality) because hyperproperties cannot specify which traces are allowed for every single context. On the other hand, it is possible that FAC does not preserve (the robust satisfaction of) hyperproperties, because contextual equivalence may not capture some hyperproperty such as noninterference, as shown in Example 2. So, what kind of hyperproperties a FAC compiler is guaranteed to preserve? If P robustly satisfies H (possibly not captured by $\approx$), what is the hyperproperty that is robustly satisfied by $[\![\mathsf{P}]\!]$ for $[\![\cdot]\!]$ being FAC?

We answer this question by defining a map $\tilde{\tau} : \wp(\wp(\mathsf{Trace_S})) \rightarrow \wp(\wp(\mathbf{Trace_T}))$ so that FAC implies $\mathsf{RHP}^{\tilde{\tau}}$. The map $\tilde{\tau}$ enjoys an optimality condition making it the *best possible* description of the target guarantee for programs compiled by a FAC compiler.

Theorem 1. *If* $[\![\cdot]\!]$ *is* FAC*, then there exists a map* $\tilde{\tau}$ *such that* $[\![\cdot]\!]$ *is* $\mathsf{RHP}^{\tilde{\tau}}$*. Moreover,* $\tilde{\tau}$ *is the smallest (pointwise) with this property.*

To avoid any misunderstanding, we stress the fact that, akin to [32, Theorem 1], neither the existence, nor the optimality of $\tilde{\tau}$ can be used to argue that a FAC compiler $[\![\cdot]\!]$ is reasonably robust. The robustness of $[\![\cdot]\!]$ depends on the image of $\tilde{\tau}$ on the hyperproperties of interest: it should not be trivial, e.g., $\tilde{\tau}(\mathsf{NI_S}) = \top$ like in Example 2 nor distort the intuitive meaning of the hyperproperty itself, e.g., $\tilde{\tau}(\mathsf{NI_S})$ = "`never output 42`". In a setting in which observables are coarse enough to be common to source and target traces, i.e., $\mathsf{Traces_S} = \mathbf{Trace_T}$, it is possible to establish whether $\tilde{\tau}(H)$ has "the same meaning" as H:

Corollary 1. *If* $[\![\cdot]\!]$ *is* FAC*, then for every hyperproperty* H*,* $[\![\cdot]\!]$ *preserves the robust satisfaction of* H *iff* $\tilde{\tau}(H) \subseteq H$*, where* $\tilde{\tau}$ *is the map from Theorem 1.*

The rigorous definition of $\tilde{\tau}$ and the proof of Theorem 1 and Corollary 1 can be found in the online appendix [5]. Here, we only mention that the definition of $\tilde{\tau}$ requires information on the compiler itself, thus it can be unfeasible to compute and assess the meaningfulness of $\tilde{\tau}(H)$. Corollary 1 partially mitigates this problem by allowing to approximate $\tilde{\tau}(H)$ rather than computing it, e.g., by showing an intermediate K such that $\tilde{\tau}(H) \subseteq K \subseteq H$. We leave as future work any approximation techniques for $\tilde{\tau}$ that would make substantial use of Corollary 1.

To overcome the issues highlighted above, we extend the categorical approach to secure compilation of Tsampas et al. [48] and propose an abstract criterion that implies both FAC and RHP^{τ} for a τ defined via co-induction and therefore independent of the compiler. In Sect. 4 we shall summarize the underlying theory before introducing our criterion in Sect. 5.

4 Secure Compilation, Categorically

The basis of our approach and that of Tsampas et al. [48] is the framework of *Mathematical Operational Semantics* (MOS) [50]. Here, we briefly explain how MOS gives a mathematical description of programming languages as well as (secure) compilers and show how our earlier Example 2 fits such a framework. We refer the interested reader to the seminal paper of Turi and Plotkin [50] and the excellent introductory material of Klin [24] for more details. Further examples and applications can be found in the literature [48,49,51].

4.1 Distributive Laws and Operational Semantics

The main idea of MOS is that the semantics of programming languages, or systems in general, can be formally described through distributive laws (i.e., natural transformations of varying complexity) of a *syntax functor* Σ over a *behavior functor* B in a suitable category (in our case the category **Set** of sets and total functions [24]). The functor $\Sigma : \mathbf{Set} \to \mathbf{Set}$ represents the algebraic signature of the language and thus acts as an abstract description of its syntax. Instead, the functor $B : \mathbf{Set} \to \mathbf{Set}$ describes the behavior of the language in terms of its observable events (e.g., the behavior of a non-deterministic labeled transition system can be modeled by the functor $BX = \wp(X)^{\Delta}$, where Δ is a set of trace labels [52]);

Recall now the languages Source and **Target** of Example 2. The syntax functor for Source for a set of terms X builds terms $\Sigma\ X$ according to (the sum of all) the constructors of the language:

$$\Sigma\ X \triangleq \top \uplus (\mathbb{N} \times E) \uplus (X \times X) \uplus (E \times X),$$

where E is the set of arithmetic expressions. The behavior functor for Source is a map that for an arbitrary set X, updates a store $s \in S$, and either terminates ($\checkmark$) or returns another term in X, possibly recording that some high-variable has been modified ($\mathcal{H}$):

$$\mathsf{B}\ X \triangleq (S \times (\text{Maybe } \mathcal{H}) \times (X \uplus \checkmark))^S.$$

In **Target**, the syntax functor is $\boldsymbol{\Sigma}\ X = \Sigma\ X \uplus X$, where the extra occurrence of X corresponds to the target context $\lceil\cdot\rceil$, and $\mathbf{B}\ X \triangleq (S \times (\text{Maybe } (\mathcal{H} \uplus \mathbf{!})) \times (X \uplus \checkmark))^S$. We explicitly notice that syntactic "holes" are represented by the identity functor Id $X = X$ and, to make this connection clearer, the syntax functor for Source can be equivalently written as $\Sigma \triangleq \top \uplus (\mathbb{N} \times E) \uplus (\text{Id} \times \text{Id}) \uplus (E \times \text{Id})$.

Next, we can define the operational semantics, a *distributive law* of Σ over B, in the format of a *GSOS law* ([24, Section 6.3]). A GSOS law of Σ over B is a natural transformation $\rho : \Sigma(\text{Id} \times B) \Longrightarrow B\Sigma^*$, where Σ^* is the free monad over Σ. For instance, the rules of sequential composition in Source (see seq1 and seq2 in the online appendix [5, Fig. 4]) correspond to the following component of the GSOS law $\rho : \Sigma(\text{Id} \times \mathsf{B}) \Longrightarrow \mathsf{B}\Sigma^*$:

$$(\mathsf{p}, \mathsf{f})\ ;\ (\mathsf{q}, \mathsf{g}) \mapsto \lambda\ s. \begin{cases} (s', \delta, \mathsf{p}'\ ;\ \mathsf{q}) & \text{if } \mathsf{f}(s) = (s', \delta, \mathsf{p}') \\ (s', \delta, \mathsf{q}) & \text{if } \mathsf{f}(s) = (s', \delta, \checkmark) \end{cases}$$

Here, $\mathsf{p}, \mathsf{q} \in X$ with X a generic set of terms, i.e., p and q can be programs, contexts or programs within a context, and $\mathsf{f}, \mathsf{g} \in \mathsf{B}X$. The image of ρ is an element of $\mathsf{B}\Sigma^* X = (S \times (\text{Maybe } \mathcal{H}) \times (\Sigma^* X \uplus \checkmark)))^S$, depending on whether p transitions to a term p' (thus involving seq2), or terminates with $\checkmark$ (seq1).

Lastly, we informally recall that when the formal semantics of a language is given through a GSOS law $\rho : \Sigma(\text{Id} \times B) \Longrightarrow B\Sigma^*$, for $\Sigma, B : \mathbf{Set} \to \mathbf{Set}$, the set of programs is (isomorphic to) the initial algebra $A = \Sigma^*\emptyset$, while the final coalgebra $Z = B^\infty\top$[2] describes the set of all possible behaviors.

Remark 2. A distributive law ρ induces a map $f : A \to Z$ that assigns to every closed term or program its behaviors as specified by the law ρ itself.

For Source and **Target** from Example 2 f and **f** are just another, equivalent representation of $\text{behs}_{\mathsf{S}}(\cdot)$ and $\mathbf{beh}_{\mathbf{T}}(\cdot)$, e.g., for v private variable,

$$\mathsf{f}([\mathtt{v}{:=}42]) = \lambda s.\ \langle s_{[x \leftarrow 42]}, \langle \mathcal{H}, \checkmark \rangle\rangle$$
$$\mathbf{f}(\lceil \mathtt{v}{:=}42 \rceil) = \lambda s.\ \langle s_{[x \leftarrow 42]}, \langle \mathbf{!}, \checkmark \rangle\rangle$$

In other words, map $f : A \to Z$ is the abstract counterpart of map *beh*$(\cdot)$ that assigns to every program the set of all its possible execution traces.

[2] Σ^* is the *free monad* over Σ and B^∞ is the *co-free comonad* over B [22, Ch. 5].

4.2 Maps of Distributive Laws as Fully Abstract Compilers

Watanabe [51] first introduced maps of distributive laws (MoDL) as *well-behaved translations* between two GSOS languages. Tsampas et al. [48] showed how MoDL can also be used as a formal, abstract criterion for secure compilation. Let us recall the definition of MoDL for two GSOS laws in the same category.

Definition 6 (MoDL). *A map of distributive law between* $\rho : \Sigma(\mathrm{Id} \times \mathsf{B}) \Longrightarrow \mathsf{B}\Sigma^*$ *and* $\boldsymbol{\rho} : \boldsymbol{\Sigma}(\mathrm{Id}\times\mathbf{B}) \Longrightarrow \mathbf{B}\boldsymbol{\Sigma}^*$ *is a pair of natural transformations* $s : \Sigma \Longrightarrow \boldsymbol{\Sigma}^*$ *and* $b : \mathsf{B} \Longrightarrow \mathbf{B}$ *such that the following diagram commutes,*

$$\begin{tikzcd}
\Sigma(\mathrm{Id} \times \mathsf{B}) \arrow[r, "\rho"] \arrow[d, "s^* \circ \Sigma(id \times b)"'] & \mathsf{B}\Sigma^* \arrow[d, "b \circ \mathsf{B} s^*"] \\
\boldsymbol{\Sigma}(\mathrm{Id} \times \mathbf{B}) \arrow[r, "\boldsymbol{\rho}"] & \mathbf{B}\boldsymbol{\Sigma}^*
\end{tikzcd}$$

where $s^* : \Sigma^* \Longrightarrow \boldsymbol{\Sigma}^*$ *extends* $s : \Sigma \Longrightarrow \boldsymbol{\Sigma}^*$ *to a morphism of free monads, i.e., to terms of arbitrary depth via structural induction.*

The diagram in Definition 6 expresses a form of *compatibility* of the source and the target semantics. Considering any source term, executing it w.r.t. the source semantics ρ and then translating the behavior (together with the resulting source term) is equivalent to first compiling the source term (and translating the behavior of its subterms) and then executing it w.r.t. the target semantics $\boldsymbol{\rho}$.

We recall that the set of source (resp. target) programs is $\mathsf{A} \triangleq \Sigma^*\emptyset$ ($\mathbf{A} \triangleq \boldsymbol{\Sigma}^*\emptyset$ resp.), and that $[\![\cdot]\!] \triangleq s^*_\emptyset : \mathsf{A} \to \mathbf{A}$ is the *compiler* induced by s. On the behaviors side, the natural transformation $b : \mathsf{B} \Longrightarrow \mathbf{B}$ induces a translation of behaviors $d := b^\infty_\top : \mathsf{Z} \to \mathbf{Z}$ where $Z \triangleq B^\infty\top$. The compiler $[\![\cdot]\!] = s^*_\emptyset$ preserves (and also reflects when all the components of b are injective) *bisimilarity* (see [48], Section 4.3). Whenever bisimilarity coincides with trace equality (see Definition 4), for example under the assumption of *determinacy*[3], the following holds ([48]).

Corollary 2. *In absence of internal non-determinism, MoDL implies* FAC.

Similarly to FAC, the definition of MoDL does not ensure that $[\![\cdot]\!] = s^*_\emptyset$ is robust. Indeed, the obvious embedding compiler from Example 2 is a MoDL (let $s = i_1$ and $b = (S \times (\mathrm{Maybe}\ i_1) \times (1 \uplus \checkmark))^S$). Intuitively, MoDL adequately captures the fact that compilation preserves the behavior of terms, but fails to capture the observations target contexts can make on compiled terms.

5 Reconciling Fully Abstract and Robust Compilation

To account for the shortcoming of MoDL, we introduce a new, complementary definition that allows reasoning explicitly on the semantic power of contexts in

[3] It is possible to eliminate the hypothesis of determinacy when B is an endofunctor over categories richer that **Set**, e.g., **Rel** the category of sets and relations.

some target language relative to contexts in a source language. This definition acts (in conjunction with MoDL) as an abstract criterion of robust compilers.

For the new definition, we elect to qualify some constructors in Σ as *contexts* constructors so that $\Sigma \triangleq \mathfrak{C} \uplus \mathfrak{P}$ where $\mathfrak{C}$ defines the constructors for contexts and $\mathfrak{P}$ for all the rest. We also assume that the GSOS law $\rho : \Sigma(\mathrm{Id} \times B) \Longrightarrow \Sigma^* B$ respects this "logical partition" of Σ in that $\rho = [B\ i_1 \circ \rho_1,\ \rho_2]$ where $\rho_1 : \mathfrak{C}(\mathrm{Id} \times B) \Longrightarrow B\mathfrak{C}^*$ and $\rho_2 : \mathfrak{P}(\mathrm{Id} \times B) \Longrightarrow B\Sigma^*$.

Definition 7 (MMoDL). *A many layers map of distributive laws (MMoDL) between* $\rho : \mathsf{\Sigma}(\mathrm{Id} \times \mathsf{B}) \Longrightarrow \mathsf{B}\mathsf{\Sigma}^*$ *and* $\rho : \boldsymbol{\Sigma}(\mathrm{Id} \times \mathbf{B}) \Longrightarrow \mathbf{B}\boldsymbol{\Sigma}^*$ *is given by natural transformations* $b : \mathsf{B} \Longrightarrow \mathbf{B}$ *and* $t : \mathfrak{C} \Longrightarrow \mathfrak{C}^*$ *making the following commute:*

$$\begin{tikzcd}
\mathfrak{C}\mathsf{\Sigma}(\mathrm{Id} \times \mathsf{B}) \arrow[r, "\mathfrak{C}^*(\Sigma\pi_1{,}\rho)"] \arrow[d, "\mathfrak{C}^*(\Sigma\pi_1{,}\rho)"] & \mathfrak{C}(\mathrm{Id} \times \mathsf{B})\mathsf{\Sigma}^* \arrow[r, "t"] & \mathfrak{C}^*(\mathrm{Id} \times \mathsf{B})\mathsf{\Sigma}^* \arrow[r, "\rho_1"] & \mathsf{B}\mathfrak{C}^*\mathsf{\Sigma}^* \arrow[d, "b"] \\
\mathfrak{C}(\mathrm{Id} \times \mathsf{B})\mathsf{\Sigma}^* \arrow[r, "\mathfrak{C}(\mathrm{Id}\times b)"] & \mathfrak{C}(\mathrm{Id} \times \mathbf{B})\mathsf{\Sigma}^* \arrow[r, "\rho_1"] & \mathbf{B}\mathfrak{C}^*\mathsf{\Sigma}^* \arrow[r, "\mathbf{B}t^*"] & \mathbf{B}\mathfrak{C}^*\mathsf{\Sigma}^*
\end{tikzcd}$$

The top-left object, $\mathfrak{C}\mathsf{\Sigma}(\mathrm{Id} \times \mathsf{B})$, represents a target context which is filled with some source term, whose subterms exhibit some source behavior. In both paths, the plugged source terms are initially evaluated w.r.t. the source semantics. On the upper path, we first *back-translate* [16] the target context using t, then we run the resulting program w.r.t. the source semantics (ρ_1), and finally we translate the resulting behavior back to the target via b. Instead, in the lower path we first translate the resulting behavior through $\mathfrak{C}(\mathrm{Id} \times b)$, then we let the target context observe (ρ_1), and finally we back-translate *the behavior* via $\mathbf{B}t^*$.

To relate MMoDL with RHP^τ, we formulate the latter in the framework of MOS. Recall (see Remark 1) that RHP^τ holds if there exists a *back-translation* map bk that for every target context $\mathbf{C_T}$ and program P, produces a source context $bk(\mathbf{C_T}, \mathsf{P}) = \mathsf{C_s}$ such that $\mathbf{beh_T}(\mathbf{C_T}\,[\![\![\mathsf{P}]\!]\!]) = \tau(\mathsf{beh_S}(\mathsf{C_s}\,[\mathsf{P}]))$.

Remark 3 ((Abstract) RHP^τ*).* For $\tau : \mathsf{Z} \to \mathbf{Z}$, a compiler $[\![\cdot]\!]$ is RHP^τ iff there exists bk such that

$$\tau \circ \mathsf{f} \circ \mathsf{plug} \circ bk = \mathbf{f} \circ \mathbf{plug} \circ id \times [\![\cdot]\!],$$

where $f : A \to Z$ associates to every program its behaviors as specified by ρ (see Remark 2) and *plug* is the operation of plugging a term into a context.

We are now ready to state our second contribution, namely that the pairing of a MoDL (s, b) and a MMoDL (t, b) gives an (abstract) RHP^τ compiler.

Theorem 2 (MMoDL imply RHP^τ**).** *Let* $s : \mathsf{\Sigma} \Longrightarrow \boldsymbol{\Sigma}^*$, $b : \mathsf{B} \Longrightarrow \mathbf{B}$ *and* $t : \mathfrak{C} \Longrightarrow \mathfrak{C}$ *such that* (s, b) *and* (t, b) *are (respectively) a MoDL and a MMoDL from* $\rho : \mathsf{\Sigma}(\mathrm{Id} \times \mathsf{B}) \Longrightarrow \mathsf{B}\mathsf{\Sigma}^*$ *to* $\rho : \boldsymbol{\Sigma}(\mathrm{Id} \times \mathbf{B}) \Longrightarrow \mathbf{B}\boldsymbol{\Sigma}^*$*. The compiler* $[\![\cdot]\!] = s^*_\emptyset$ *is (abstract)* RHP^τ *for* $\tau = b^\infty_\top$ *coinductively induced by* b*.*

Proof (Sketch). The back-translation $bk := t^*_\emptyset \times id$ satisfies the equation in Remark 3 (details in the online appendix [5]). $\square$

Before fixing the compiler from Example 2 to make it satisfy both MoDL (Definition 6) and MMoDL (Definition 7), let us see why the back-translation mapping both target contexts to the identity source context $[\cdot]$ is not a MMoDL. Let $\mathtt{v} \in Var_H$ be a private variable, on the upper path of Definition 7 we have

$$\lceil \mathtt{v}{:=}42 \rceil \xrightarrow{\mathfrak{C}^*(\Sigma\pi_1,\rho)} \lceil \checkmark \rceil, \lambda s.\langle s_{[v\leftarrow 42]}, \mathcal{H}\rangle \xrightarrow{\quad t \quad} [,\checkmark], \ldots \mathcal{H} \xrightarrow{\rho_1} \checkmark, \ldots \mathcal{H} \xrightarrow{b} \checkmark, \ldots \mathcal{H}$$

Note how the identity context fails to report $\mathbf{!}$. On the lower path, we have instead

$$\lceil \mathtt{v}{:=}42 \rceil \xrightarrow{\mathfrak{C}^*(\Sigma\pi_1,\rho)} \lceil \checkmark \rceil, \lambda s.\langle s_{[v\leftarrow 42]}, \mathcal{H}\rangle \xrightarrow{\mathfrak{C}(\mathrm{Id}\times b)} \lceil \checkmark \rceil, \ldots \mathcal{H} \xrightarrow{\rho_1} \checkmark, \ldots \mathbf{!} \xrightarrow{\mathbf{B}t^*} \checkmark, \ldots \mathbf{!}$$

Here, it is evident that the context $\lceil \cdot \rceil$ "picks up" $\mathcal{H}$ and reports $\mathbf{!}$, unlike $[\cdot]$.

Example 3 (Example 2, revisited). We now show how to fix the compiler from Example 2 by making it RHP^τ for a suitable τ. For that, we first need to slightly modify the language **Target**. The idea is that variable assignments in **Target** should now be *sandboxed*, so that the interactions with the context $\lceil \cdot \rceil$ do not expose sensitive information. Formally, we extend the algebraic signature of **Target** with a constructor for sandboxing assignments, i.e., $\mathbf{\Sigma} \uplus (E \times \mathrm{Id})$, so that **Target** terms are generated by grammar

$$<\mathbf{P}> ::= \mathbf{skip} | \mathtt{v} := <expr> | \langle \mathbf{P} \rangle ; \langle \mathbf{P} \rangle | \mathbf{while} \ <expr> \langle \mathbf{P} \rangle | \lfloor \mathtt{v} := <expr> \rfloor$$

where the semantics of $\lfloor \cdot \rfloor$ is described in Fig. 3. We can now define the new (i.e., fixed) compiler $[\![\cdot]\!]$ and the appropriate map τ, so that $[\![\cdot]\!]$ is RHP^τ. Both $[\![\cdot]\!]$ and τ are determined by the natural transformations s, t, and b, such that (s, b) is a MoDL and (t, b) is a MMoDL. The natural transformation $s : \Sigma \Longrightarrow (\mathbf{\Sigma} \uplus (E \times \mathrm{Id}))^*$, and therefore the inductively defined compiler $[\![\cdot]\!] \triangleq s^*_\emptyset$, wraps assignments in the sandbox $\lfloor \cdot \rfloor$, i.e., $[\![\mathtt{v} := \mathtt{e}]\!] = \lfloor \mathtt{v} := \mathtt{e} \rfloor$ and acts as the identity on other terms. The natural transformation $t : \mathfrak{C} \Longrightarrow \mathfrak{C}^*$ maps every **Target** context to the identity context $[\cdot]$. Finally, the translation of behaviors $b : \mathsf{B} \Longrightarrow \mathbf{B}$ erases the occurrences of $\mathcal{H}$, implying that the compiled terms are not expected to report changes in high variables.

$$\mathbf{sb1}\ \frac{s, \mathbf{p} \xrightarrow{\mathcal{H}} s', \checkmark}{s, \lfloor \mathbf{p} \rfloor \to s', \checkmark} \qquad \mathbf{sb2}\ \frac{s, \mathbf{p} \xrightarrow{\mathcal{H}} s', \mathbf{p}'}{s, \lfloor \mathbf{p} \rfloor \longrightarrow s', \mathbf{p}'}$$

Fig. 3. Rules extending the semantics of **Target**.

Recall that the diagram from Definition 7 failed to commute for Example 2, because (s, b) being a MoDL imposed b to not erase any occurrences of $\mathcal{H}$. The same diagram for the new **Target** language and natural transformations s, b, and t now commutes. More specifically, in the upper path we have

$$\lceil \mathtt{v}{:=}42 \rceil \xrightarrow{\mathfrak{C}^*(\Sigma\pi_1,\rho)} \lceil \checkmark \rceil, \lambda s.\langle s_{[v\leftarrow 42]}, \mathcal{H}\rangle \xrightarrow{t} [,\checkmark], \ldots \mathcal{H} \xrightarrow{\rho_1} \checkmark, \ldots \mathcal{H} \xrightarrow{b} \checkmark, \ldots$$

while in the lower path we get

$$\lceil \mathtt{v}{:=}42 \rceil \xrightarrow{\mathfrak{C}^*(\Sigma\pi_1,\rho)} \lceil \checkmark \rceil, \lambda s.\langle s_{[v\leftarrow 42]}, \mathcal{H}\rangle \xrightarrow{\mathfrak{C}(\mathrm{Id}\times b)} \lceil \checkmark \rceil, \ldots \xrightarrow{\rho_1} \checkmark, \ldots \xrightarrow{\mathbf{B}t^*} \checkmark, \ldots$$

We point the reader interested to the online appendix [5] for more details in showing that the above (s, b) is a MoDL and that (t, b) is a MMoDL.

Hereafter, we discuss one of the benefits of the abstract definitions presented so far, namely that we can easily compute τ, and immediately establish if programs that robustly satisfy $\mathsf{NI_S}$ (noninterference in Source) are compiled to programs that robustly satisfy $\mathbf{NI_T}$. In order to do so, we need to connect Z and $\mathbf{Z}$ to traces and hyperproperties of Source and **Target**. Elements of Z are functions that assign to every $s \in S$ a new state s' and *maybe* an extra symbol like **!** or $\mathcal{H}$, and a continuation, i.e., another function of the same type. Traces are instead sequences of stores possibly exhibiting the extra symbols $\mathcal{H}$ and **!**. It is easy to show (see the online appendix [5]) that every trace corresponds to an element of Z – the function that returns the head of the trace and continues as the tail of the same trace – and that every function in Z corresponds to a set of traces – one trace for every fixed $s \in S$. Thus, we can prove that τ maps (the set of functions in Z corresponding to) $\mathsf{NI_S}$ to a subset of (the set of functions corresponding to) $\mathbf{NI_T}$, i.e., the compiler $[\![\cdot]\!]$ preserves robust satisfaction. ■

6 Related Work

In this section, we discuss related work regarding origins and applications of full abstraction, trace based criteria, MoDL and relevant proof techniques.

Full abstraction was introduced to relate the operational and the denotational semantics of programming languages [40]. A denotational semantics of a language is said to be fully abstract w.r.t. an operational one for the same language *iff* the same denotation is given to contextually equivalent terms, i.e., those terms that result the *same* when evaluated according to the operational semantics. Common choices to establish when the result of the evaluation is *the same*, and hence to define contextual equivalence, are *equi-convergence* and *equi-divergence* (e.g., in [13,14,23,28,38]). Notice that there is no loss of generality with these choices, if (and only if!) contexts are powerful enough [28], e.g., when all inputs can be thought as part of the context, and the context itself may select one final value as the result of the execution or diverge.

Fully abstract translations as in Definition 1 have been adopted for comparing expressiveness of languages (see, e.g., the works by Mitchell [28] and Patrignani et al. [38]), but Gorla and Nestmann [21] showed that they may lead to false positive results. The interested reader can find out more in the online appendix [5], where we also sketch how to use RHP^τ for expressiveness comparisons.

Full Abstraction and Secure Compilation. Abadi [1] originally proposed to use full abstraction to preserve security properties in translations from a source language L_1 to a target one L_2. A fully abstract translation or compiler preserves and reflects equivalences, and can therefore be a way to preserve security properties when these are expressed as equivalences. Remarkable examples from the literature are given by Bowman and Ahmed [13], Busi et al. [14] and Skorstengaard et al. [43]. In the first two works the authors model contexts so that contextual equivalence captures (forms of) noninterference and preserve it through a fully abstract translation. Skorstengaard et al. [43] consider a source language with well-bracketed control flow (WBCF) and local state encapsulation (LSE), then model target contexts so that these two properties are captured by contextual equivalence and, they exhibit a fully abstract translation so that both WBCF and LSE are guaranteed also in the target. We stress the fact that, all security properties that are not captured by contextual equivalence are not necessarily preserved by a fully abstract compiler, thus allowing for counterexamples similar to Example 1. Finally, it is worth noting that fully abstract compilation does not prevent source programs to be insecure, nor suggests how to fix them, quoting Abadi [1]:

> An expression of the source language L_1 may be written in a silly, incompetent, or even malicious way. For example, the expression may be a program that broadcasts some sensitive information—so this expression is insecure on its own, even before any translation to L_2. Thus, full abstraction is clearly not sufficient for security [...]

Beyond Full Abstraction. Several definitions of "well-behaved translations" exist, depending both on the scenario and on the properties one aims to preserve during the translation. For example, if the goal is to preserve functional correctness, then it is natural to require the compiled program to *simulate* the source one [29]. This can be expressed both as a relation between the operational semantics of the source and the target (see for example [27,42,51]), or extrinsically as a relation between the execution traces of programs before and after compilation [3,12,46]. *Trace based criteria for compiler correctness* The CompCert [12,26] and CakeML [46] projects are milestones in the formal verification of compilers. Preservation of functional correctness can be expressed in both cases in terms of execution traces [3]. For the CompCert compiler, executing $[\![\mathsf{P}]\!]$ w.r.t. the target semantics yields the same observable events as executing P w.r.t. the source semantics, as long as P does not encounter an undefined behavior. Similarly, CakeML ensures that executing $[\![\mathsf{P}]\!]$ w.r.t. the target semantics yields the same observable events as executing P w.r.t. the source semantics, as long as there is enough space in target memory. In both cases, correctness is proven by exhibiting a simulation between $[\![\mathsf{P}]\!]$ and P.

Trace Based Criteria for Secure Compilation. Similarly to what happens for functional correctness, relations between the execution traces of a program and of its compiled version, can be used to express preservation of noninterference through compilation [10,11,30]. The simulation-based techniques introduced in CompCert sometimes suffice also to show the preservation of noninterference, e.g., when the source and the target semantics are equipped with a notion of leakage [10, Sections 5.2–5.4]. However, in more general cases a stronger, *cube-shaped simulation* is needed (see [10, Section 5.5], and [11,30]). Stewart et al. [44] propose a variant of CompCert that also gives some guarantees w.r.t. source contexts, and their compilation in the target. Still, this does not guarantee security against *arbitrary* target contexts, that can be strictly stronger than source ones. Abate et al. [3,4] propose a family of criteria with the goal of preserving satisfaction of (classes of) security properties against arbitrary contexts. Also, they show that their criteria can be formulated in at least two equivalent ways. The first one explicitly describes the target guarantees ensured for compiled programs, for example which safety properties are guaranteed for programs written in unsafe languages and compiled according to the criterion proposed by Abate et al. [2] (see their Appendix A). The second way is instead more amenable to proofs, e.g., by enabling proofs by back-translation Abate et al. [2, Fig. 4].

Maps of Distributive Laws (MoDL). Mathematical Operational Semantics (MOS) and distributive laws ensure *well-behavedness* of the operational semantics of a language while also providing a formal description for it. Such semantics have been given for languages with algebraic effects [6] and for stochastic calculi [25]. In their biggest generality distributive laws are defined between monads and comonads [24], but it is often convenient to consider the slightly less general GSOS laws that correspond bijectively to GSOS rules [7,24,41].

Proof Techniques for fully abstract compilation include both cross-language logical relations between source and compiled programs [13,35,43] and back-translation of target contexts into source ones [14,16,35]. The latter technique sometimes exploits information from execution traces [16], and can be adapted also to some of the robust criteria of Abate et al. [4]. Ongoing work is aiming to formalize the back-translation technique needed to prove some of the robust preservation of safety (hyper)properties in the Coq proof assistant [2,47]. The best results in mechanization of secure compilation criteria have been achieved for the criteria that can be proven via simulations, especially when extending the CompCert proof scripts, e.g., [10]. The complexity of many proofs is relatively contained as they show a *forward* simulation—the source program simulates the one in the target—and "flip" it into a *backward* one—the compiled program simulates the source one—with a general argument. We are not aware of mechanized proofs for MoDL, but we believe it would be convenient to first express maps between GSOS laws as relations between GSOS rules (see also Sect. 7).

7 Conclusions and Future Work

The scope of this work has been to clarify the guarantees provided by criteria for secure compilation, make them explicit and immediately accessible to users and developers of (provably) secure compilers. We investigated the relation between fully abstract and robust compilation, provided an explicit description of the hyperproperties robustly preserved by a fully abstract compiler, and noticed that these are not always meaningful, nor of practical utility. We have therefore introduced a novel criterion that ensures both fully abstract and robust compilation, and such that the meaningfulness of the hyperproperty guaranteed to hold after compilation can be easily established. The proposed example shows that our criterion is achievable.

Future work will focus on proof techniques for MoDL and MMoDL that are amenable to formalization in a proof assistant. For that we can either build on existing formalizations of polynomial functors as containers, or exploit the correspondence between GSOS laws and GSOS rules, and characterize MoDL and MMoDL as relations between source and target rules. Another interesting line of work consists in devising over (under) approximation for the map $\tilde{\tau}$ from Theorem 1, and use our Corollary 1 to establish whether existing fully abstract compilers preserve (violate) a given hyperproperty.

Acknowledgements. We are grateful to Pierpaolo Degano, Letterio Galletta, Catalin Hritcu, Marco Patrignani, Frank Piessens, and Jeremy Thibault for their feedback on early versions of this paper. We would also like to thank the anonymous reviewers for their insightful comments and suggestions that helped to improve our presentation.

Carmine Abate is supported by the European Research Council https://erc.europa.eu/ under ERC Starting Grant SECOMP (715753). Matteo Busi is partially supported by the research grant on *Formal Methods and Techniques for Secure Compilation* from the Department of Computer Science of the University of Pisa.

References

1. Abadi, M.: Protection in programming-language translations. In: Secure Internet Programming, Security Issues for Mobile and Distributed Objects, pp. 19–34 (1999)
2. Abate, C., et al.: When good components go bad: formally secure compilation despite dynamic compromise. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 1351–1368 (2018)
3. Abate, C., et al.: Trace-relating compiler correctness and secure compilation. In: ESOP 2020. LNCS, vol. 12075, pp. 1–28. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-44914-8_1
4. Abate, C., Blanco, R., Garg, D., Hritcu, C., Patrignani, M., Thibault, J.: Journey beyond full abstraction: exploring robust property preservation for secure compilation. In: 2019 IEEE 32nd Computer Security Foundations Symposium (CSF), pp. 256–25615. IEEE (2019)
5. Abate, C., Busi, M., Tsampas, S.: Fully abstract and robust compilation and how to reconcile the two, abstractly (2021)
6. Abou-Saleh, F., Pattinson, D.: Towards effects in mathematical operational semantics. Electr. Notes Theor. Comput. Sci. **276**, 81–104 (2011)
7. Aceto, L., Fokkink, W., Verhoef, C.: Structural operational semantics. In: Handbook of Process Algebra, pp. 197–292. Elsevier (2001)
8. Ahmed, A., Blume, M.: Typed closure conversion preserves observational equivalence. In: Hook, J., Thiemann, P. (eds.) Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming, ICFP 2008, Victoria, BC, Canada, 20–28 September 2008, pp. 157–168. ACM (2008)
9. Ahmed, A., Blume, M.: An equivalence-preserving CPS translation via multi-language semantics. In: Chakravarty, M.M.T., Hu, Z., Danvy, O. (eds.) Proceeding of the 16th ACM SIGPLAN International Conference on Functional Programming, ICFP 2011, Tokyo, Japan, 19–21 September 2011, pp. 431–444. ACM (2011). https://doi.org/10.1145/2034773.2034830
10. Barthe, G., et al.: Formal verification of a constant-time preserving C compiler. Proc. ACM Program. Lang. **4**(Popl), 1–30 (2019)

11. Barthe, G., Grégoire, B., Laporte, V.: Secure compilation of side-channel countermeasures: the case of cryptographic "constant-time". In: 2018 IEEE 31st Computer Security Foundations Symposium (CSF), pp. 328–343. IEEE (2018)
12. Besson, F., Blazy, S., Wilke, P.: A verified compcert front-end for a memory model supporting pointer arithmetic and uninitialised data. J. Autom. Reason. **62**(4), 433–480 (2019)
13. Bowman, W.J., Ahmed, A.: Noninterference for free. ACM SIGPLAN Not. **50**(9), 101–113 (2015)
14. Busi, M., et al.: Provably secure isolation for interruptible enclaved execution on small microprocessors. In: 33rd IEEE Computer Security Foundations Symposium (CSF 2020) (2020)
15. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. **18**(6), 1157–1210 (2010)
16. Devriese, D., Patrignani, M., Piessens, F.: Fully-abstract compilation by approximate back-translation. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 164–177 (2016)
17. D'Silva, V., Payer, M., Song, D.X.: The correctness-security gap in compiler optimization. In: 2015 IEEE Symposium on Security and Privacy Workshops, SPW 2015, San Jose, CA, USA, 21–22 May 2015, pp. 73–87 (2015)
18. Durumeric, Z., et al.: The matter of heartbleed. In: Proceedings of the 2014 Conference on Internet Measurement Conference, pp. 475–488 (2014)
19. El-Korashy, A., Tsampas, S., Patrignani, M., Devriese, D., Garg, D., Piessens, F.: CapablePtrs: securely compiling partial programs using the pointers-as-capabilities principle. CoRR abs/2005.05944 (2020)
20. Engelfriet, J.: Determinacy - (observation equivalence = trace equivalence). Theor. Comput. Sci. **36**(1), 21–25 (1985)
21. Gorla, D., Nestmann, U.: Full abstraction for expressiveness: history, myths and facts. Math. Struct. Comput. Sci. **26**(4), 639–654 (2016)
22. Jacobs, B.: Introduction to Coalgebra: Towards Mathematics of States and Observation, Cambridge Tracts in Theoretical Computer Science, vol. 59. Cambridge University Press (2016). ISBN 9781316823187
23. Jacobs, K., Timany, A., Devriese, D.: Fully abstract from static to gradual. Proc. ACM Program. Lang. **5**(Popl), 1–30 (2021)
24. Klin, B.: Bialgebras for structural operational semantics: an introduction. Theoret. Comput. Sci. **412**(38), 5043–5069 (2011)
25. Klin, B., Sassone, V.: Structural operational semantics for stochastic process calculi. In: Amadio, R. (ed.) FoSSaCS 2008. LNCS, vol. 4962, pp. 428–442. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78499-9_30
26. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: Morrisett, J.G., Jones, S.L.P. (eds.) Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, 11–13 January 2006, pp. 42–54. ACM (2006)
27. Melton, A., Schmidt, D.A., Strecker, G.E.: Galois connections and computer science applications. In: Pitt, D., Abramsky, S., Poigné, A., Rydeheard, D. (eds.) Category Theory and Computer Programming. LNCS, vol. 240, pp. 299–312. Springer, Heidelberg (1986). https://doi.org/10.1007/3-540-17162-2_130
28. Mitchell, J.C.: On abstraction and the expressive power of programming languages. Sci. Comput. Program. **21**(2), 141–163 (1993)

29. Morris, F.L.: Advice on structuring compilers and proving them correct. In: Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 144–152 (1973)
30. Murray, T., Sison, R., Pierzchalski, E., Rizkallah, C.: Compositional verification and refinement of concurrent value-dependent noninterference. In: 2016 IEEE 29th Computer Security Foundations Symposium (CSF), pp. 417–431. IEEE (2016)
31. Nielson, H.R., Nielson, F.: Semantics with Applications: An Appetizer. Undergraduate Topics in Computer Science. Springer. London (2007). https://doi.org/10.1007/978-1-84628-692-6. ISBN 978-1-84628-691-9
32. Parrow, J.: General conditions for full abstraction. Math. Struct. Comput. Sci. **26**(4), 655–657 (2016)
33. Patrignani, M.: Why should anyone use colours? Or, syntax highlighting beyond code snippets. CoRR abs/2001.11334 (2020)
34. Patrignani, M., Agten, P., Strackx, R., Jacobs, B., Clarke, D., Piessens, F.: Secure compilation to protected module architectures. ACM Trans. Program. Lang. Syst. **37**(2), 6:1–6:50 (2015)
35. Patrignani, M., Ahmed, A., Clarke, D.: Formal approaches to secure compilation: a survey of fully abstract compilation and related work. ACM Comput. Surv. (CSUR) **51**(6), 1–36 (2019)
36. Patrignani, M., Clarke, D.: Fully abstract trace semantics for protected module architectures. Comput. Lang. Syst. Struct. **42**, 22–45 (2015)
37. Patrignani, M., Garg, D.: Secure compilation and hyperproperty preservation. In: 2017 IEEE 30th Computer Security Foundations Symposium (CSF), pp. 392–404. IEEE (2017)
38. Patrignani, M., Martin, E.M., Devriese, D.: On the semantic expressiveness of recursive types. Proc. ACM Program. Lang. **5**(Popl), 1–29 (2021)
39. Piessens, F.: Security across abstraction layers: old and new examples. In: 2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW), pp. 271–279. IEEE (2020)
40. Plotkin, G.D.: LCF considered as a programming language. Theoret. Comput. Sci. **5**(3), 223–255 (1977)
41. Plotkin, G.D.: A structural approach to operational semantics. Aarhus university (1981)
42. Sabry, A., Wadler, P.: A reflection on call-by-value. ACM Trans. Program. Lang. Syst. (TOPLAS) **19**(6), 916–941 (1997)
43. Skorstengaard, L., Devriese, D., Birkedal, L.: StkTokens: enforcing well-bracketed control flow and stack encapsulation using linear capabilities. Proc. ACM Program. Lang. **3**(Popl), 19:1–19:28 (2019)
44. Stewart, G., Beringer, L., Cuellar, S., Appel, A.W.: Compositional compcert. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 275–287 (2015)
45. Strydonck, T.V., Piessens, F., Devriese, D.: Linear capabilities for fully abstract compilation of separation-logic-verified code. Proc. ACM Program. Lang. **3**(ICFP), 84:1–84:29 (2019)
46. Tan, Y.K., Myreen, M.O., Kumar, R., Fox, A., Owens, S., Norrish, M.: The verified cakeML compiler backend. J. Func. Program. **29** (2019)
47. Thibault, J., Hritcu, C.: Nanopass back-translation of multiple traces for secure compilation proofs. In: 5th Workshop on Principles of Secure Compilation, PriSC 2021, Virtual Event, 17 January 2021 (2021). http://perso.eleves.ens-rennes.fr/people/Jeremy.Thibault/prisc2021.pdf

48. Tsampas, S., Nuyts, A., Devriese, D., Piessens, F.: A categorical approach to secure compilation. In: Petrişan, D., Rot, J. (eds.) CMCS 2020. LNCS, vol. 12094, pp. 155–179. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-57201-3_9
49. Turi, D.: Categorical modelling of structural operational rules: case studies. In: Category Theory and Computer Science, 7th International Conference, CTCS 1997, Santa Margherita Ligure, Italy, 4–6 September 1997, Proceedings, pp. 127–146 (1997)
50. Turi, D., Plotkin, G.: Towards a mathematical operational semantics. In: Proceedings of Twelfth Annual IEEE Symposium on Logic in Computer Science, pp. 280–291. IEEE (1997)
51. Watanabe, H.: Well-behaved translations between structural operational semantics. Electr. Notes Theoret. Comput. Sci. **65**(1), 337–357 (2002)
52. Winskel, G., Nielsen, M.: Models for concurrency. DAIMI Rep. Ser. **22**(463) (1993)