Journal of Logical and Algebraic Methods in Programming 125 (2022) 100736

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

A formal treatment of the role of verified compilers in secure computation

José Carlos Bacelar Almeida [a], Manuel Barbosa [b,*], Gilles Barthe [c,d], Hugo Pacheco [b], Vitor Pereira [e,1], Bernardo Portela [b]

[a] Universidade do Minho and INESC TEC, Portugal
[b] University of Porto (FCUP) and INESC TEC, Portugal
[c] MPI for Security and Privacy, Germany
[d] IMDEA Software Institute, Spain
[e] SRI International, United States of America

ARTICLE INFO

Article history:
Received 30 March 2020
Received in revised form 14 November 2021
Accepted 15 November 2021
Available online 19 November 2021

Keywords:
Secure multiparty computation
Secure compilation
Certified compilation
Formal verification
EasyCrypt
Computer-aided cryptography

ABSTRACT

Secure multiparty computation (SMC) allows for complex computations over encrypted data. Privacy concerns for cloud applications makes this a highly desired technology and recent performance improvements show that it is practical. To make SMC accessible to non-experts and empower its use in varied applications, many domain-specific compilers are being proposed.

We review the role of these compilers and provide a formal treatment of the core steps that they perform to bridge the abstraction gap between high-level ideal specifications and efficient SMC protocols. Our abstract framework bridges this secure compilation problem across two dimensions: 1) language-based *source*- to *target*-level semantic and efficiency gaps, and 2) cryptographic *ideal*- to *real*-world security gaps. We link the former to the setting of certified compilation, paving the way to leverage long-run efforts such as CompCert in future SMC compilers. Security is framed in the standard cryptographic sense. Our results are supported by a machine-checked formalisation carried out in EasyCrypt.

1. Introduction

Secure multiparty computation (SMC) is a cryptographic technology that enables mutually distrusting parties to jointly perform computations while retaining control over their secret inputs. A solid theoretical foundation supports SMC, establishing feasibility and impossibility results for general secure computation over many dimensions of the design space, including the number of parties, the class of computations, the trust model and the power of attackers.

In the last decade, efficient protocols have been developed and optimised for specific design goals (concrete computations, restricted classes of attackers, etc.). These protocols were successfully deployed in a number of real-world scenarios [1,2]. In this paper we consider a class of SMC protocols that, due to its efficiency and generality, has received significant attention from the academic community and industry alike [3]. These protocols are built on top of linear secret sharing, which underlies several of the state-of-the-art SMC frameworks in the literature, such as Sharemind [4], SCALE-MAMBA [5], SPDZ [6], FRESCO [7], PICCO [8] or JIFF [9].

* Corresponding author.
E-mail address: mbb@fc.up.pt (M. Barbosa).

[1] Part of this work was developed while the author was at the University of Porto (FCUP) and INESC TEC.

https://doi.org/10.1016/j.jlamp.2021.100736
2352-2208/

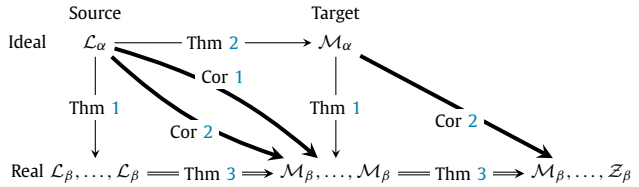


Fig. 1. Compilation dimensions as performed by domain-specific SMC frameworks.

We consider core transformations that such frameworks must deal with, as depicted in Fig. 1, annotated with our results for further reference. The overarching goal is to allow (possibly non-expert) developers to program particular applications via a high-level specification language $\mathcal{L}$ that hides the low-level details of the secure computation protocols underneath. At this level, public and secret data are seen as inhabitants of data types in a high-level language; computations over secret data are idealised/modularised as calls to an *ideal* static library α—visible at the top left of Fig. 1—as if they were performed in the clear by a trusted third-party (TTP). SMC tool-chains usually provide implementations of the semantics of these ideal-world libraries that developers can use to animate and debug their high-level computation specifications.

The role of the framework is then to compile such idealised specifications into framework-specific executable code for the various parties. This code will orchestrate calls to a distributed static library β that implements low-level SMC protocols for each of the operations idealised in α, and thus allow the different parties to collaboratively carry out the computation over secret-shared data. Conceptually, such frameworks are carrying out two types of transformations: 1) replacing the semantic domain of secret data from idealised TTP-like processing α to real-world secret-shared data processing β; and 2) taking a program in high-level language $\mathcal{L}$ and transforming it into a collection of communicating programs that the various parties can execute, either in the same low-level language $\mathcal{M}$, or in different low-level languages $\mathcal{N} \ldots \mathcal{Z}$. The final outcome is visible at the bottom right of Fig. 1.

The tool-chains in existing SMC frameworks are crafted to demonstrate cryptographic advances, such as a new SMC protocol, or novel program analysis or transformations, such as a type system or a code optimisation; a recent literature review [10] frames this area of research as follows:

> *Programming languages are a field dedicated to creating compilers but little SMC research leverages these techniques. [...] However, the SMC community would benefit if frameworks took a more principled approach to language design and verification.*

In this paper, we argue that the way to fill this gap is to split the problem of secure compilation for SMC across the two dimensions shown in Fig. 1, and ask the following question:

> *Can one decompose the problem of secure compilation for SMC into two orthogonal dimensions—source-to-target language semantic gap and ideal-to-real-world security gap—so that compilation tools can be analysed independently for each type of transformation?*

We frame this question by proposing a language-based execution model that exposes this duality (Section 2). Here, ideal computations are expressed as the evaluation of a program under the semantics of a programming language and make calls to an external static library; this static library defines the low-level computations that can be performed securely using (atomic) SMC protocols. This view of ideal functionalities is fully general, and may be of independent interest.

We then formalise a framework for program-based secure computation (Section 3) that can be parameterised with an arbitrary secure computation library. We adopt a notion of secure SMC libraries known as an Arithmetic Black Box [11] (ABB), in that the secure computation library is idealised as a reactive general-purpose computer that receives inputs and may keep internal state, but only makes the outputs publicly known. This allows for intermediate operations computed inside the ABB to have a weaker form of security than that of the full reactive functionality, that must be secure in a standard Universal Composability [12] (UC) sense. To make these notions concrete, our formalisation is inspired by Sharemind [4], one of several frameworks that implements an ABB, and in which intermediate protocols satisfy a concrete weaker form of security known as *privacy*.[2] This allows our framework to preserve the strong composition guarantees ensured by protocols shown to be UC secure. Moreover, our execution model implicitly assumes global synchronisation points between computing parties when executing low-level protocols. In this first step we also limit our attention to passive security and static corruptions.

We answer the above question by showing how a secure compilation framework may follow any path in Fig. 1 to guarantee the security of the generated protocols. Along the way, we identify sufficient conditions on the components of the framework that clearly expose the duality between the programming languages (horizontal) and the cryptographic security domains (vertical):

[2] Note that relaxing the security requirement on low-level protocols only strengthens the result.

- Vertically, we prove that, under natural assumptions on the language and the static libraries, a secure compiler that replicates the ideal program to all computing parties and replaces the ideal static library with a secure distributed one is guaranteed to produce a UC-secure SMC protocol in the presence of passive adversaries and static corruptions (Section 4).
- Horizontally, we prove that the trace-preservation properties offered by general-purpose certified compilers such as CompCert [13] are sufficient to make our diagram commute, by either changing the language in the ideal world, from $\mathcal{L}_\alpha$ to $\mathcal{M}_\alpha$, or in the real world, where parties may independently compile their local copies of $\mathcal{L}_\beta$ to $\mathcal{M}_\beta, \ldots, \mathcal{Z}_\beta$ (Section 5).

Our methodology was formalised and machine-checked in EasyCrypt, an interactive theorem prover specialising in code-based cryptographic security proofs. Our EasyCrypt development, including a detailed description of our proof, is publicly available at https://github.com/haslab/LangSMC. A significant part of our formalisation effort is precisely concerned with unifying notions from the cryptography and programming languages domains, of which we will present a high-level EasyCrypt description along the paper. In particular, we reason about reactive cryptographic functionalities against passive adversaries (conceptualised in the general UC framework [12]), and how these can be realised as the semantics of interactive programming languages. Our approach to formalising UC for reactive functionalities and passive adversaries in EasyCrypt is novel, as is the formalisation of interactive programming languages and their semantics using EasyCrypt's inductive data types.

Section 6 identifies promising research directions and finishes with some closing remarks. Appendix A provides a complete reference of the notation presented throughout the paper. Appendix B discusses related work, and in particular how concrete real-world SMC frameworks fit into our general program-based framework.

2. Programming languages and semantics

Our framework is parametric on, i.e., abstracts the details of, programming languages and their semantics. Therefore, we stress that our secure compilation results are postulated generically for any language (that satisfies certain assumptions) and support a broad class of secure computation frameworks, easing future realisation efforts.

One such assumption, which greatly clarifies presentation, is to have the framework delegating all secret data manipulation to an external library accessed through what we call a security API. This library will not only be responsible for performing operations on secret data, but also for storing the secret values inside its internal state. That would mean that programs in the programming language only manipulate public handlers (or locations) of secret values.

Relation to realistic SMC compilers. This semantic distinction between the orchestration language and the secure computation library leaves much flexibility to the kinds of secure operations supported, and is completely agnostic to the concrete security enforcement mechanisms that a framework may employ to guarantee that programs respect a security specification. Intuitively, this is to say that our framework assumes a pre-processing of user-level programs that statically identifies which parts will be run securely and slices them into public and secret counterparts. In realistic SMC compilers, such slicing can take various forms. Some compilers assume that the whole program is run securely [14]. Typically, the user-level language features a security type system that syntactically distinguishes all operations manipulating secure types to be run securely [4,5,8,15]. Additionally, the mapping of secure operations in the user-level language to secure operations in the library may be mostly straightforward [4,5], or involve compiling arbitrary user-level programs (classically when secure programs can branch on secret data) to programs (without branching) that can be run obliviously [14,15,8]. Optimising compilers may also automatically infer secure operations from a security specification [16] (e.g., given secret inputs and public outputs, infer which intermediate operations need to be run securely), or go further and have a syntactic separation of public and secret data but support multiple security domains and automatically infer in which security domain to run each secure operation [17]. We stress that our framework is orthogonal to such pre-processing of user-level programs and this is precisely what enables us to abstract over the languages and secure computation libraries.

2.1. Programming language

A language $\mathcal{L}$ defines a syntax for programs, and a corresponding a small-step semantics presented as a binary relation $\rightarrow_{\mathcal{L}}$ on configurations (referred as local configurations). A local configuration ($\sigma = \langle P, \rho, \zeta \rangle$) comprises the program under evaluation (P), an environment (ρ) that keeps track of the program local state and a call state (ζ) that mediates the execution of the external secure computation API that will handle operations on secret data. A call state can be empty ($\bot$), a call to a secure operation ($\mathsf{Call}_{\mathsf{sop}}(\ldots)$), or the return of a call ($\mathsf{Ret}(\ldots)$). We will always assume the existence of two distinguished secure operations for I/O (input and output), allowing us to capture the typical behaviour of reactive secure computation functionalities and protocols.

Note that ζ achieves two goals: on one hand, it matches the typical architecture of SMC frameworks, which factors out low-level operations over encrypted data into some form of static library or virtual machine; on the other hand, it creates a formal boundary in our models that permits looking at language semantics as being parametric on the semantic domain for secret data. Intuitively, in the ideal-world, calls in ζ specify the functionality of cryptographic protocols; in the real-world, calls in ζ will trigger the execution of such distributed cryptographic protocols that cannot be modelled within the language. The boundary between ζ and the language semantics unifies these domains by seeing them as interaction events mediated by in the call state.

$$e ::= x \mid v \mid \mathsf{pop}\,(\vec{e})$$

$$\mathsf{sop} ::= \mathsf{input} \mid \mathsf{output} \mid \ldots$$

$$P ::= \mathsf{skip} \mid x := e \mid P_1; P_2 \mid \mathsf{if}\ e\ \mathsf{then}\ \{P_1\}\ \mathsf{else}\ \{P_2\} \mid \mathsf{while}\ e\ \mathsf{do}\ \{P\}$$
$$\mid x_\perp := \mathsf{sop}\,(\vec{e_1}; \vec{e_2}; e_{3\perp})$$

$$\frac{}{\langle \mathsf{skip}\,; P, \rho, \perp\rangle \to_{\mathcal{L}} \langle P, \rho, \perp\rangle} \qquad \frac{\rho(e) = v}{\langle x := e, \rho, \perp\rangle \to_{\mathcal{L}} \langle \mathsf{skip}\,, \rho[x \mapsto v], \perp\rangle}$$

$$\frac{\langle P_1, \rho, \perp\rangle \to_{\mathcal{L}} \langle P_1', \rho', \perp\rangle}{\langle P_1; P_2, \rho, \perp\rangle \to_{\mathcal{L}} \langle P_1'; P_2, \rho', \perp\rangle}$$

$$\frac{\rho(e) = v \qquad v \neq 0}{\langle \mathsf{if}\ e\ \mathsf{then}\ \{P_1\}\ \mathsf{else}\ \{P_2\}, \rho, \perp\rangle \to_{\mathcal{L}} \langle P_1, \rho, \perp\rangle} \qquad \frac{\rho(e) = v \qquad v = 0}{\langle \mathsf{if}\ e\ \mathsf{then}\ \{P_1\}\ \mathsf{else}\ \{P_2\}, \rho, \perp\rangle \to_{\mathcal{L}} \langle P_2, \rho, \perp\rangle}$$

$$\frac{\rho(\vec{e_1}) = \vec{v_1} \quad \rho(\vec{e_2}) = \vec{v_2} \quad \rho(e_3) = v_3}{\langle x := \mathsf{sop}\,(\vec{e_1}; \vec{e_2}; e_3), \rho, \perp\rangle \to_{\mathcal{L}} \langle x := \mathsf{sop}\,(\vec{e_1}; \vec{e_2}; e_3), \rho, \mathsf{Call}_{\mathsf{sop}}(\vec{v_1}; \vec{v_2}; x; v_3)\rangle}$$

$$\frac{\rho' = [x \mapsto v] \text{ if } x \neq \perp, \text{ else } \rho}{\langle x := \mathsf{sop}\,(\vec{e_1}; \vec{e_2}; e_3), \rho, \mathsf{Ret}(v)\rangle \to_{\mathcal{L}} \langle \mathsf{skip}\,, \rho', \perp\rangle}$$

Fig. 2. A simple security-aware interactive while language.

As we need to evaluate a program $P \in \mathcal{L}$ by different entities (parties) in a coherent manner, we shall rely on two main assumptions on its semantics:

- Deterministic local reduction: if $\sigma_1 \to_{\mathcal{L}} \sigma_2$ and $\sigma_1 \to_{\mathcal{L}} \sigma_3$, then $\sigma_2 = \sigma_3$;
- Block on calls: if $\sigma = \langle P, \rho, \mathsf{Call}_{\mathsf{sop}}(\ldots)\rangle$, then $\sigma \not\to_{\mathcal{L}}$.

The rationale behind these assumptions is that: i) calls must be the same for all executions of the same program; ii) calls to the external API might trigger the execution of distributed protocols, hence forcing synchronisation.[3]

EasyCrypt formalisation. It is instructive to have a look on how the abstract notion of languages have been formalised in EasyCrypt. We have modelled them as an EasyCrypt *theory*, as it allows greater flexibility in, e.g., instantiating and/or cloning. The semantics is modelled as a partial function `lstep`, hence determinism is built on it by construction. The `lcallSt` queries/extracts an API call from a local configuration, returning all information characterising it (secret operation; list of public arguments; list of handlers for secret arguments; an optional variable to be assigned with the public result; and an optional handler for the secret result). Finally, `lcallRet` processes the public result.

```
type V, sop_t.
theory Lang.
 type L, var_t, lconf_t.
 op lconf_init: L → lconf_t.
 op lstep: lconf_t → lconf_t option.
 op lcallSt: lconf_t → (sop_t*V list*V list*V option) option.
 op lcallRet: V → lconf_t → lconf_t.
 axiom block_on_calls sigma: lcallSt sigma != None ⇒ lstep sigma = None.
end Lang.
```

A concrete language. For concreteness, we present a simple imperative interactive language from [18], extended with secret operators, and considering only secret I/O channels (Fig. 2), that fits our semantic structure. We stress that our results are postulated generically, and in no way restricted to this language.

We will use the colour blue to distinguish concrete language syntax from language-agnostic constructions. In this toy language, x denotes variables and values v range over the corresponding semantic domain $\mathcal{V}$ (e.g. bounded integers). We will also use h when referring to values when used as handlers. Within expressions e, pop denotes any total n-ary operation over integers, $\vec{e}$ a sequence of expressions and $e_\perp$ an optional expression. Programs P are defined as usual by sequencing commands, among which we highlight calls to the secure API $x := \mathsf{sop}\,(\vec{e_1}; \vec{e_2}; e_{3\perp})$, denoting a call to the sop operation

[3] In fact, these assumptions could be relaxed: i) a program may have non-deterministic reductions as long as it has a deterministic call trace; ii) API operations with no communication may support a local evaluation strategy in the distributed setting. For simplicity of presentation we have chosen not to follow a more refined treatment.

$$\alpha\text{-IN}\ \frac{\theta' = \langle \phi[h_{\text{ho}} \mapsto \mathsf{unshare}(\bar{v}_{\text{in}})], \bot, \mathcal{O}\rangle}{(\theta', \emptyset, \mathcal{C}(\bar{v}_{\text{in}})) \leftarrow \mathit{EvalSop}^{\alpha}_{\mathsf{input}}(\langle \phi, \bar{v}_{\text{in}}, \mathcal{O}\rangle, [], [], h_{\text{ho}})}$$

$$\alpha\text{-OUT}\ \frac{\bar{v}_{\text{out}} \leftarrow \mathsf{share}(\phi[h_{\text{out}}]) \quad \theta' = \langle \phi, \mathcal{I}, \bar{v}_{\text{out}}\rangle}{(\theta', \emptyset, \mathcal{C}(\bar{v}_{\text{out}})) \leftarrow \mathit{EvalSop}^{\alpha}_{\mathsf{output}}(\langle \phi, \mathcal{I}, \bot\rangle, [], [h_{\text{out}}], \bot)}$$

$$\alpha\text{-SOP}\ \frac{\begin{array}{c}\mathsf{sop} \notin \{\mathsf{input}, \mathsf{output}\} \quad \mathrm{ar}(\mathsf{sop}) = (|\vec{v}_{\text{pi}}|, |\vec{h}_{\text{hi}}|, h_{\text{ho}} \neq \bot) \\ (v_{\text{ho}}, l) = \mathcal{F}_{\mathsf{sop}}(\vec{v}_{\text{pi}}, \phi[\vec{h}_{\text{hi}}]) \quad \theta' = \langle \phi[h_{\text{ho}} \mapsto v_{\text{ho}}], \mathcal{I}, \mathcal{O}\rangle\end{array}}{(\theta', \mathsf{pubres}_{\mathsf{sop}}(l), l) \leftarrow \mathit{EvalSop}^{\alpha}_{\mathsf{sop}}(\langle \phi, \mathcal{I}, \mathcal{O}\rangle, \vec{v}_{\text{pi}}, \vec{h}_{\text{hi}}, h_{\text{ho}})}$$

Fig. 3. Evaluation of secret operations in α.

with public parameters $\vec{e}_1$, handlers of secret inputs $\vec{e}_2$, and storing the secret result in handler e_3 ($\bot$ if it doesn't apply). The public result is assigned to variable x (or not assigned if $\bot$). The corresponding semantic rule sets the call state, hence blocking computation.[4]

The general syntax for API calls blurs the reactive nature of the language. As a convenience, one might consider statements input e and output e as shorthands of respectively $\bot$:=input([]; []; e) and $\bot$:=output([]; [e]; $\bot$). Here we see that input expects a single secret output handler (where the input from the environment will be stored), and output a single secret input handler (whose content shall be sent to the environment). These are the only two operations assumed in our setting, but for the sake of illustration we describe a couple of commonly found operations on secret data that might be included in the API: x:=declassify([]; [e]; $\bot$) that reveals a secret value with handler e; $\bot$:=add([]; [e_1, e_2]; e_3) that adds secret values of e_1 and e_2 and stores it in e_3; $\bot$:=scalarmul([e_1]; [e_2]; e_3) that multiplies a public value e_1 by the secret value of e_2 and stores it in e_3; x:=new_svar([]; []; $\bot$) that generates a new handler to hold secret values, stored in variable x, useful in allowing the API to get stricter control of the secret store.

2.2. Computation over secret data

Let us turn our attention to the external secret computation library, accessed through an API, and for which we will provide two different implementations: a so called ideal-world instance α, that acts as the specification of the corresponding functionality, and a real-world counterpart β, that will capture the execution of the distributed cryptographic protocols securely implementing the aforementioned functionality. In both implementations of the API, the set of secret operations sop is fixed, and we also assume a fixed arity function $\mathrm{ar} : \mathsf{sop} \to \mathcal{N} \times \mathcal{N} \times \mathcal{B}$, that describes for each operation the number of public and secret arguments, and whether it assigns a secret output.

An implementation of the API keeps in its internal state ($\theta = \langle \phi, \mathcal{I}, \mathcal{O}\rangle$) a secret store ($\phi$), mapping handles in $\mathcal{V}$ to a semantic domain of secret values, and input ($\mathcal{I}$) and output ($\mathcal{O}$) buffers, both holding an optional secret-shared value ($\bar{\mathcal{V}}_{\bot}$), that mediate I/O with the external environment. In the ideal-world implementation α, the semantic domain of secret values is identified with the value domain $\mathcal{V}$ (secrets are kept on the clear). In the real-world, we assume a secret-sharing scheme that splits secret values among different parties. We denote values in secret-shared form as $\bar{v} \in \bar{\mathcal{V}}$, and the randomised procedure that converts a value in its shared form by share (being unshare its left inverse). The evaluation of a secret operation sop is described by a randomised procedure $\mathit{EvalSop}_{\mathsf{sop}}$. We write $(\theta', v_{\text{po}}, \tau) \leftarrow \mathit{EvalSop}_{\mathsf{sop}}(\theta, \vec{v}_{\text{pi}}, \vec{h}_{\text{hi}}, h_{\text{ho}})$ to mean that evaluating operation sop with internal state of the API θ, public inputs $\vec{v}_{\text{pi}}$, secret input handles $\vec{h}_{\text{hi}}$, and the optional output handle h_{ho}, produces the updated internal state θ', the public result v_{po}, and additional side-information (we will often use $l \in \mathcal{T}$ for functionalities and $\tau \in \mathcal{T}$ for protocols), which use shall become clear later.

Fig. 3 details the conditions enforced upon evaluation of secret operations in the ideal-world library α. For operations other than input and output, it assumes the existence of specification functions $\mathcal{F}_{\mathsf{sop}} : \mathcal{V}^{\star} \times \mathcal{V}^{\star} \to \mathcal{V} \times \mathcal{T}$ and $\mathsf{pubres}_{\mathsf{sop}} : \mathcal{T} \to \mathcal{V}$. Function $\mathcal{F}_{\mathsf{sop}}(\vec{v}_{\text{pi}}, \vec{v}_{\text{hi}})$ specifies the secret result and *leakage* $l \in \mathcal{T}$ to be released as side-information. Function $\mathsf{pubres}_{\mathsf{sop}}$ specifies the public result from the previously specified leakage (or some default value $\emptyset \in \mathcal{V}$ if meaningful), hence enforcing its public nature. Rule α-SOP essentially performs the required manipulations on the secret store. Rules α-IN and α-OUT additionally retrieve/set the input and output buffers, and release as side information those shares associated to corrupted parties (function $\mathcal{C}(-)$), since they are supposed to be available to the adversary – see Section 4.

Regarding the evaluation of secret operations in the real-world library β (Fig. 4), we note that in Rule β-SOP the specifications $\mathcal{F}_{\mathsf{sop}}$ are replaced by protocol descriptions π_{sop}. These are modelled as randomised procedures returning the shared secret output, and as side information τ the communication trace containing all messages sent/received to/from corrupted parties. Additionally, we also expect the operator's leakage to be reconstructible from the generated trace τ (as $\mathsf{leak}^{\pi}_{\mathsf{sop}}(\tau)$), thus attesting its public nature.

[4] The notations for environment lookup ($\rho(e)$) and update ($\rho[x \mapsto v]$) are extended for sequences and optional expressions/variables.

$$\beta\text{-IN}\ \frac{\tau \leftarrow \pi_{\mathsf{input}}(\bar{v}_{\mathrm{in}}) \quad \theta' = \langle \phi[h_{\mathrm{ho}} \mapsto \bar{v}_{\mathrm{in}}], \bot, \mathcal{O}\rangle}{(\theta', \emptyset, \tau) \leftarrow \mathit{EvalSop}^{\beta}_{\mathsf{input}}(\langle \phi, \bar{v}_{\mathrm{in}}, \mathcal{O}\rangle, [], [], h_{\mathrm{ho}})}$$

$$\beta\text{-OUT}\ \frac{(\bar{v}_{\mathrm{out}}, \tau) \leftarrow \pi_{\mathsf{output}}(\phi[h_{\mathrm{ho}}]) \quad \theta' = \langle \phi, \mathcal{I}, \bar{v}_{\mathrm{out}}\rangle}{(\theta', \emptyset, \tau) \leftarrow \mathit{EvalSop}^{\beta}_{\mathsf{output}}(\langle \phi, \mathcal{I}, \bot\rangle, [], [h_{\mathrm{ho}}], \bot)}$$

$$\beta\text{-SOP}\ \frac{\begin{array}{c}\mathsf{sop} \notin \{\mathsf{input}, \mathsf{output}\} \quad \mathrm{ar}(\mathsf{sop}) = (|\vec{v}_{\mathrm{pi}}|, |\vec{v}_{\mathrm{hi}}|, v_{\mathrm{ho}} \neq \bot) \\ (\bar{v}_{\mathrm{ho}}, \tau) \leftarrow \pi_{\mathsf{sop}}(\vec{v}_{\mathrm{pi}}, \phi[\vec{h}_{\mathrm{hi}}]) \\ \theta' = \langle \phi[h_{\mathrm{ho}} \mapsto \bar{v}_{\mathrm{ho}}], \mathcal{I}, \mathcal{O}\rangle\end{array}}{(\theta', \mathsf{pubres}_{\mathsf{sop}}(\mathsf{leak}^{\pi}_{\mathsf{sop}}(\tau)), \tau) \leftarrow \mathit{EvalSop}^{\beta}_{\mathsf{sop}}(\langle \phi, \mathcal{I}, \mathcal{O}\rangle, \vec{v}_{\mathrm{pi}}, \vec{h}_{\mathrm{hi}}, h_{\mathrm{ho}})}$$

Fig. 4. Evaluation of secret operations in β.

$$\textsc{IdealSem-step-local}\ \frac{\sigma \rightarrow_{\mathcal{L}} \sigma'}{\langle \sigma, \theta\rangle \overset{\epsilon}{\Rightarrow} \langle \sigma', \theta\rangle}$$

$$\textsc{IdealSem-step-call}\ \frac{(\theta', v_{\mathrm{po}}, l) \leftarrow \mathit{EvalSop}^{\alpha}_{\mathsf{sop}}(\theta, \vec{v}_1, \vec{h}_2, h_3)}{\langle\langle P, \rho, \mathsf{Call}_{\mathsf{sop}}(\vec{v}_1, \vec{h}_2, x, h_3)\rangle, \theta\rangle \overset{l}{\Rightarrow} \langle\langle P, \rho[x \mapsto v_{\mathrm{po}}], \bot\rangle, \theta'\rangle}$$

Fig. 5. Ideal-world semantics.

We also note that rules β-IN and β-OUT include now protocols π_{input} and π_{output} respectively. While the former can be as simple as forwarding shares to each party, the later will consist of a distributed resharing protocol, as will be detailed later (Section 4).[5]

EasyCrypt formalisation. Libraries α and β are modelled directly as EasyCrypt modules, that encapsulate both an internal state and probabilistic imperative procedures. The interface of the API is captured by the following module type:

```
module type API_t = {
  proc init(): unit
  proc eval_sop(o:sop_t, pargs sargs:V list, sres: V option): V option * SideInfo
  proc set_input(x: S): bool
  proc get_output(): S option
}.
module Alpha: API_t = { ... (*\alpha implementation*) }
module Beta: API_t = { ... (*\beta implementation*) }
```

Module implementations `Alpha` and `Beta` (implementing respectively α and β) keep their internal state θ, initialised by the `init` method. The `eval_sop` method implements rules of Fig. 3 in `Alpha`, and Fig. 4 in `Beta`. Finally, `set_input` fills the input buffer if empty (returning `false` if already full), and `get_output` collects data and clears the output buffer.

2.3. Ideal- and real-world semantics

Plugging together the language semantics and the libraries implementing the security API allow us to characterise both the idealised functionality (ideal-word) and the concrete distributed implementation (real-word).

The ideal-world scenario is modelled by a single party executing the program $P \in \mathcal{L}$, and resorting to the library α on calls to the security API. Since α performs all computation on secret data in the clear, the overall effect is thus that the combined semantics act as the specification of the intended functionality. The semantics is given as a relation $\Rightarrow$, defined over global configurations that combine both a local configuration σ and the internal state θ of the API, and is tagged with side-information l, τ representing the side-information released by the evaluation of secret operations (ϵ being the empty side-information). It is defined by the rules presented in Fig. 5. Rule IdealSem-step-local lifts local reduction to the ideal-world semantics, and rule IdealSem-step-call performs the required adjustments to the call-state of the local configuration upon execution of an API call.

For the real-world semantics, a set of parties $\mathcal{P}$ jointly computes the functionality. Each party locally executes its own copy of a program P (or possibly different related programs), cooperating through the execution of protocols triggered by calls to the real-world library β. As we want to model each party advancing asynchronously, and synchronising only where required by the distributed protocols (that is, in calls to the API), we keep distinct relations for asynchronous (local) steps, denoted by $\overset{i}{\rightarrow}$ where $i \in \mathcal{P}$, and synchronous (distributed) steps denoted by $\overset{\tau}{\Rightarrow}$ as in α. The real-world semantic rules are presented in Fig. 6. Configurations in the real-world semantics combine an indexed set of local configurations ($\Sigma = [\sigma^i]_{i \in \mathcal{P}}$)

[5] These protocols are tailored to the class of SMC protocols that we consider in our framework, and may be adapted for other classes of SMC protocols.

$$\textsc{RealSem-stepP}\ \frac{\Sigma(i) = \sigma \quad \sigma \to_{\mathcal{L}} \sigma'}{\langle \Sigma, \theta \rangle \xrightarrow{i} \langle \Sigma[i \mapsto \sigma'], \theta \rangle}$$

$$\textsc{RealSem-stepS}\ \frac{(\theta', v_{\mathrm{po}}, \tau) \leftarrow \mathit{EvalSop}^{\beta}_{\mathsf{sop}}(\theta, \vec{v}_1, \vec{h}_2, h_3)}{\left\langle \left[\langle P^i, \rho^i, \mathsf{Call}_{sop}(\vec{v}_1, \vec{h}_2, x, h3)\rangle\right]_{i\in\mathcal{P}}, \theta \right\rangle \overset{\tau}{\Longrightarrow} \left\langle \left[\langle P^i, \rho[x \mapsto v_{\mathrm{po}}]^i, \bot\rangle\right]_{i\in\mathcal{P}}, \theta' \right\rangle}$$

Fig. 6. Real-world semantics.

with the internal state of the external library β. We write $\Sigma(i)$ when referring to the local configuration of party i, and $\Sigma[i \mapsto \sigma]$ to the update of local configuration of party i in Σ. Rule RealSem-stepP equates $\xrightarrow{i}$ to the local reduction $\to_{\mathcal{L}}$ of the underlying language running at party i. Rule RealSem-stepS enforces synchronisation by requiring that the call state of all local configurations is the same. After evaluation of sop in β, the public result is passed back to local configurations, and the communication trace τ exposed.

EasyCrypt formalisation. Modules `IdealSem` and `RealSem` implement the ideal and real-world semantics respectively. They implement the following interface:

```
module type IdealSem_t = {
  proc init(P: L): unit
  proc step(): SideInfo option
}
module IdealSem : IdealSem_t = { ... }
module RealSem_t = {
  proc init(P: L): unit
  proc stepP(i: int): bool
  proc stepS(): SideInfo option
}
module RealSem: RealSem_t = { ... }
```

The local configurations are stored in the internal state of modules `IdealSem` and `RealSem`, and initialised by method `init`. The other methods encode step relations, that explicitly return whether they succeed.

Multiple programming languages. For the sake of readability, we have presented the real-world semantics based on multiple parties running the same program. But it is worth mentioning that the definition makes perfect sense with multiple programs possibly from different programming languages. Assuming that each party $i \in \mathcal{P}$ runs a program from language $\mathcal{L}_i$, a refined rule for the public-step relation of the real-world semantics would be:

$$\textsc{RealSem-stepP(multiL)}\ \frac{\Sigma(i) = \sigma \quad \sigma \to_{\mathcal{L}_i} \sigma'}{\langle \Sigma, \theta \rangle \xrightarrow{i} \langle \Sigma[i \mapsto \sigma'], \theta \rangle}$$

Since the API is shared between all parties, and independent, no adjustment is needed to the rule RealSem-stepS.

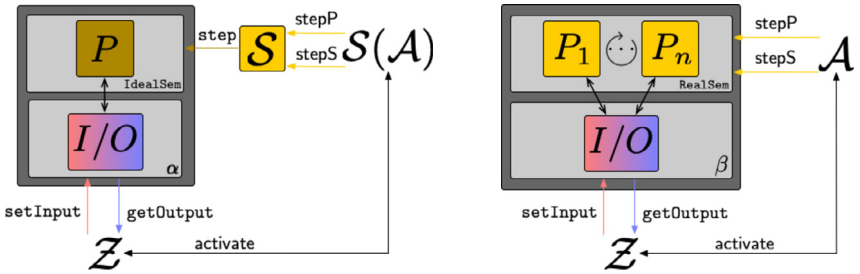
A question that arises is if it makes sense for different parties to execute different programs. Indeed, it should now be clear that synchronisation at API calls imposes a tight coherence among parties, which we shall see in Section 5 that is it precisely what we get from certified compilation.

3. Program-based secure computation

This section establishes a unified formal framework in which one can simultaneously reason about program transformations and cryptographic protocol security. In the UC framework [12], cryptographic security of SMC protocols is defined using a strong variant of the simulation paradigm. Intuitively, no attacker shall distinguish a *real* world from an *ideal* world, even when this attacker is collaborating with an adversarial environment that can control inputs and observe outputs produced by the protocol. In the real world, an attacker interacts directly with the multi-party protocol execution according to a set of rules that define the attack model. In the ideal world, the attacker interacts with an ideal functionality that emulates the role of the parties as if the protocol was executed by a TTP. A protocol is secure if there exists a simulator that can emulate the real-world view observable by the attacker, while interacting with the ideal functionality. Composability of the UC security notion relies crucially on the fact that the simulation must be consistent even when the attacker is in collusion with other adversarial entities that may control the environment in which the protocol is executed.

3.1. Security model

Our framework (Fig. 7) considers two external entities: an *environment* $\mathcal{Z}$ and an *adversary* $\mathcal{A}$, that collude while interacting with the system. In the real world, a set of parties $\mathcal{P}$ locally execute a program that orchestrates co-ordinated



(a) Ideal world. (b) Real world.

Fig. 7. Security interfaces in our program-based secure computation model.

execution of the SMC protocol and, when needed, engage in distributed protocols with other parties using secure channels. We allow for a subset of parties $\mathcal{C} \subset \mathcal{P}$ to be *corrupt*, sharing their internal state and messages sent/received to the adversary $\mathcal{A}$. The set $\mathcal{C}$ is statically defined, i.e. it is fixed at the beginning of the protocol.[6] The adversary is *passive*, in the sense that it does not actively intervene in the distributed protocols, but we let it control the pace that each party adopts when executing its local computation. This is because we want our security notion to be independent of "how" each party performs its local computation.

In the ideal world the ideal functionality is running internally a single copy of a reference program that specifies the distributed computation to be carried out. Computation on secret data is *hidden* inside the functionality itself according to given specifications $\mathcal{F}_{\mathsf{sop}}$; furthermore, the functionality also specifies *leakage* to be revealed to the simulator (e.g. I/O exchanged with corrupted parties, size information, etc.). This leakage should be interpreted as information that the protocol is *not* required to hide from the attacker.

Informally, security is established when no environment $\mathcal{Z}$ is capable of distinguishing which side of Fig. 7 is interacting with the system (for any adversary $\mathcal{A}$). Stated differently, there exists a *simulator* $\mathcal{S}$ that, interacting with the ideal world, is able to reconstruct an indistinguishable adversarial view of the real world. Formally we will use a notion of perfect indistinguishability, i.e., identity of distributions which is natural in EasyCrypt. The whole formalisation could be generalised to rely on statistical or computational indistinguishability at the cost of an overhead in verification work and loss in readability.

Adversarial interface. Looking at the interactions between each component of Fig. 7, we observe that programs in both worlds progress based on *step* commands delivered via the adversarial interface accessible to $\mathcal{A}$ and $\mathcal{S}$. The stepS interface in the real-world reveals communications traces observable by the corrupt parties that may occur in distributed computations; alternatively the adversary may drive a single party to progress in its local computations via stepP. In the ideal world, a single step method is made available, possibly revealing some leakage; this allows the simulator to control the pace at which the ideal computation progresses, e.g., so that the outputs produced by parties match the timing at which they occur in the real world. In either world, the environment can activate the adversary whenever it wishes to trigger progress of parties. Additionally, the environment can also control the input and output buffers interacting with the reactive semantics, through the procedures `getInput` and `setOutput`.

EasyCrypt formalisation. The interactions from Fig. 7 are better understood by looking directly at the formalisation. We first note that adversarial entities in EasyCrypt are modelled as *abstract modules*, i.e. modules with an interface, but which are not instantiated. When these entities have access to oracles, they are passed to their interface as parameters. Hence, a model of the environment $\mathcal{Z}$, with oracle access to the methods mentioned above, can be constructed with the following EasyCrypt code:

```
module type Z_IO_t = {
  proc set_input(x: S): bool
  proc get_output(): S option
}.
type any.
module type Z_Adv_t = {
  proc activate(): any
}.
module type Z_t(Z_IO: Z_IO_t, Z_Adv: Z_Adv_t) = {
  proc run(): bool
}.
```

[6] This permits simplifying our formalisation to assume from the start that the execution traces produced by low-level protocols contain only the values observable by the corrupt parties, cf., real-word security library β in Section 2.2.

An environment is an abstract module with module type `Z_t`. It has a single procedure `run` that produces a boolean – it can be thought as its guess whether is the interacting with the real or the ideal world. Since it is parameterised by `Z_IO_t` and `Z_Adv_t`, it will have access to oracles allowing it to call the methods `set_input`, `get_output` and `activate`. The first two are supposed to interact with the semantics (I/O buffers), and the latter activates the adversary so that it can make progress on parties at his will. A note on the result type of the `activate` method – we let it be an abstract type, saying nothing about it. Although abstract, it is probably simpler to think of the type of information passed to the environment from the adversary as the aggregation of all information collected by the adversary while interacting with the semantics.

The reasoning followed for specifying the environment can also be adopted for the adversary and the simulator interfaces:

```
(* Adversary interface *)
module type A_Sem_t = {
  proc stepP(i: int): bool
  proc stepS(): SideInfo option
}.
module type Adv_t(Sem: A_Sem_t) = {
  proc init(P:L): unit
  proc activate(): any
}.
(* Simulator interface *)
module type S_Sem_t = {
  proc step(): SideInfo option
}.
module type Sim_t(Sem: S_Sem_t) = {
  proc init(P:L): unit
  proc activate(): any
}.
```

We are now ready to construct the ideal- and real-world experiments of Fig. 7 (a.k.a. *security games*). $\mathsf{IDEAL}[\mathcal{Z},\mathcal{A}](P)$ and $\mathsf{REAL}[\mathcal{Z},\mathcal{A}](P)$.

```
module IDEAL(Z: Z_t, A: Sim_t) = {
  module Adv = A(IdealSem))
  proc game(P: L): bool = {
    var b;
    Alpha.init();
    IdealSem.init(P);
    Adv.init(P);
    b <@ Z(Alpha, Adv).run();
    return b;
  }
}.
module REAL(Z: Z_t, A: Adv_t) = {
  module Adv = A(RealSem))
  proc game(P: L): bool = {
    var b;
    Beta.init();
    RealSem.init(P);
    Adv.init(P);
    b <@ Z(Beta, Adv).run();
    return b;
  }
}.
```

Notice that the above definitions are well-typed, because the interface `Z_IO_t` is indeed a sub-interface of `API_t`. Likewise, `A_Sem_t` (respectively `S_Sem_t`) are sub-interfaces of `RealSem_t` (respectively `IdealSem_t`).

Security definition. Formally, security is defined as the existence of a simulator $\mathcal{S}$ that makes indistinguishable the games $\mathsf{REAL}[\mathcal{Z},\mathcal{A}]$ and $\mathsf{IDEAL}[\mathcal{Z},\mathcal{S}(\mathcal{A})]$. Here we adopt the multi-program view of the real-world semantics, to present the definition in its full generality (cf. discussion at the end of Section 2.3).

Definition 1 *(Program-based secure computation).* A multiparty evaluation of programs $P_1,\ldots,P_n$ with real API β is said to securely compute program P with ideal API α if there exists a simulator $\mathcal{S}$ such that, for all adversaries $\mathcal{A}$ and environments $\mathcal{Z}$,

$$\mathsf{REAL}[\mathcal{Z},\mathcal{A}](P_1,\ldots,P_n) \sim \mathsf{IDEAL}[\mathcal{Z},\mathcal{S}(\mathcal{A})](P).$$

The symbol $\sim$ denotes equivalence of the probabilistic programs describing the games REAL and IDEAL, or equivalently, equality of the boolean random variables of their outcome for the same program.

Relation to universal composability. Our program-based secure computation follows the universal composability model, refined to the concrete setting that we are analysing and syntactically adjusted to better fit the EasyCrypt proof-assistant. Our model enforces synchronisation between all parties in I/O operations. This prevents the environment from activating a party before the environment defines the inputs to all participants in a given input round, and it prevents the environment from collecting local party outputs before all parties have produced the output for an output round. This simplification is consistent with a setting where input/output is carried out by a dedicated I/O party interacting with the computing parties via authenticated channels. For example, in the input case, the environment would fix the input of all parties in shared form, and would only then be able to schedule the delivery of shares to the computing parties. We omit these details from the execution model for simplicity.

Contrarily to common practice in UC execution models, the simulator activates the functionality as necessary, rather than the other way around. In the case of the protocols we analyse this is without loss of generality, as all operations that allow the environment to observe the functionality are preceded by a secure operation, in which the simulator is activated. This facilitates the proof, as the simulator can orchestrate the execution instead of requesting information from the functionality and book-keeping program states. We also note that, currently, EasyCrypt has no means of enforcing computational bounds on simulators and adversaries. We could, however, manually prove that our constructed simulators run within polynomial time given polynomial-time low-level simulators.

Finally, the ideal functionality animates the program following a small-step semantics instead of big-step activations between I/O operations. The reasoning is the same as for the previous point: it gives more control to the simulator over the execution and reduces proof complexity related with book-keeping. One can design a big-step simulator by allowing it to control when I/O is performed, activating the functionality when necessary, following the small-step strategy, and only allowing I/O operations on the functionality when all parties are synchronised.

4. Vertical dimension: single-program secure computation

We now turn our attention to the vertical dimension of Fig. 1, that instantiates Definition 1 for the case where the ideal and real-world programs are written in the same language (called either $\mathcal{L}$ or $\mathcal{M}$), but rely on different APIs to compute over confidential data. Note that this means that, in the real world, all parties are executing the same program as the ideal functionality, in which calls to the ideal API α replaced by calls to the (distributed) real-world API β. This realisation of Definition 1 has two immediate consequences: i. the local semantics of each party in the real world is a copy of the local semantics for an ideal world; and ii. security of the overall computation will depend essentially on the security of the API implementation with which the language is linked.

Security requires the existence of a simulator that is able to mimic the behaviour of the corrupt parties that participate in the distributed evaluation of the program, using only the leakage given by a sequential execution in the ideal world. Clearly, the critical point is on the relation between APIs α and β, specifically on the security assurances offered when evaluating individual secure operations. But these elementary assumptions need to be promoted to the entire program. Intuitively, we will construct a simulator for the execution of an entire program by orchestrating the simulators that are implied by our security assumptions on the individual protocols implemented by the secure API, exploring their compositional properties to prove that same-program SMC is secure.

4.1. API security

Our setting allows for arbitrary I/O interactions with the environment. Moreover, as has been presented in Section 2.3, these interactions are made in shared form. It is thus important that, to not jeopardise security, the outputted shared values do not reveal any information used/stored internally on the library β (e.g. a program can repeatedly output the same secret, in which case the shares should not be equal, as otherwise it would be trivial to distinguish its behaviour with the ideal world where fresh shares are produced). In practice, the above problem can be circumvented by forcing the π_{output} protocol to calculate a fresh sharing of the secret, so that an adversary can learn nothing from the concrete values of its shares. The simulator $\mathcal{S}_{\mathsf{output}}$ must produce an indistinguishable corrupt communication trace from the corrupt input and output shares. Following the nomenclature of [19,20], we denote this property as *security*.

For intermediate operations, it is possible to relax the security property, particularly when in the passive scenario with static corruptions. Concretely, one may relax the need for the protocols to produce randomised outputs, improving the efficiency of the overall evaluation – *freshness* of shares is only enforced when revealing outputs to the environment. Simulators shall not only need to be able to reproduce communication traces of corrupt parties, but also to be able to reproduce the output shares of those parties from their respective inputs. This property is called *privacy* [20,19].

Summing up, in order to establish the security of library β with respect to α, we demand the existence of simulators $\mathcal{S}_{\mathsf{sop}}$ exhibiting the mentioned features.

Definition 2 *(API security).* The real library β is said to be a secure realisation of the ideal library α, if there exist simulators $\mathcal{S}_{\mathsf{output}}$, $\mathcal{S}_{\mathsf{input}}$, and $\mathcal{S}_{\mathsf{sop}}$ (for any other secret operation other that input and output), such that the experiments on the left and right hand side of Fig. 8 are indistinguishable.

$\texttt{outputSimL}(\bar{v}_{\text{sarg}})$:

1 : $\bar{v}_{\text{out}} \leftarrow \mathsf{share}(\mathsf{unshare}(\bar{v}_{\text{sarg}}))$

2 : $\tau \leftarrow \mathcal{S}_{\mathsf{output}}\,(\mathcal{C}(\bar{v}_{\text{sarg}}), \mathcal{C}(\bar{v}_{\text{out}}))$

3 : **return** $(\mathsf{unshare}(\bar{v}_{\text{sarg}}), \mathcal{C}(\bar{v}_{\text{out}}), \tau)$

$\texttt{outputSimR}(\bar{v}_{\text{sarg}})$:

1 : $(\bar{v}_{\text{out}}, \tau) \leftarrow \pi_{\mathsf{output}}\,(\bar{v}_{\text{sarg}})$

2 : **return** $(\mathsf{unshare}(\bar{v}_{\text{out}}), \mathcal{C}(\bar{v}_{\text{out}}), \tau)$

$\texttt{inputSimL}(\bar{v}_{\text{in}})$:

1 : $\tau \leftarrow \mathcal{S}_{\mathsf{input}}\,(\mathcal{C}(\bar{v}_{\text{in}}))$

2 : **return** τ

$\texttt{inputSimR}(\bar{v}_{\text{in}})$:

1 : $\tau \leftarrow \pi_{\mathsf{input}}\,(\bar{v}_{\text{in}})$

2 : **return** τ

$\texttt{sopSimL}(\vec{v}_{\text{pi}}, \vec{\bar{v}}_{\text{hi}})$:

1 : $(v_{\text{ho}}, l) \leftarrow \mathcal{F}_{\mathsf{sop}}\,(\vec{v}_{\text{pi}}, \mathsf{unshare}(\vec{\bar{v}}_{\text{hi}}))$

2 : $(\bar{v}'_{\text{ho}}, \tau) \leftarrow \mathcal{S}_{\mathsf{sop}}\,(\vec{v}_{\text{pi}}, \mathcal{C}(\vec{\bar{v}}_{\text{hi}}), l)$

3 : **return** $(v_{\text{ho}}, \bar{v}'_{\text{ho}}, l, \tau)$

$\texttt{sopSimR}(\vec{v}_{\text{pi}}, \vec{\bar{v}}_{\text{hi}})$:

1 : $(\bar{v}_{\text{ho}}, \tau) \leftarrow \pi_{\mathsf{sop}}\,(\vec{v}_{\text{pi}}, \vec{\bar{v}}_{\text{hi}})$

2 : **return** $(\mathsf{unshare}(\bar{v}_{\text{ho}}), \mathcal{C}(\bar{v}_{\text{ho}}), \mathsf{leak}^{\pi}_{\mathsf{sop}}(\tau), \tau)$

Fig. 8. Simulator experiments.

$$\gamma\text{-IN}\ \frac{\zeta(\bar{v}_{\text{in}}) = l \qquad \tau \leftarrow \mathcal{S}_{\mathsf{input}}\,(\zeta(\bar{v}_{\text{in}}))}{\phi[h_{\text{in}} \mapsto \bar{v}_{\text{in}}] \leftarrow \mathit{SimSop}^{\gamma}_{\mathsf{input}}\,(\phi, [], [], h_{\text{in}}, l)}$$

$$\gamma\text{-OUT}\ \frac{\zeta(\bar{v}_{\text{out}}) = l \qquad \tau \leftarrow \mathcal{S}_{\mathsf{output}}\,(\phi[h_{\text{ho}}], \zeta(\bar{v}_{\text{out}}))}{(\phi, \bot, \tau) \leftarrow \mathit{SimSop}^{\gamma}_{\mathsf{output}}\,(\phi, [], [h_{\text{ho}}], \bot, l)}$$

$$\gamma\text{-SOP}\ \frac{\begin{array}{c}\mathrm{ar}(\,\mathsf{sop}\,) = (|\vec{v}_{\text{pi}}|, |\vec{\bar{v}}_{\text{hi}}|, v_{\text{ho}} \neq \bot) \\ (\bar{v}_{\text{ho}}, \tau) \leftarrow \mathcal{S}_{\mathsf{sop}}\,(\vec{v}_{\text{pi}}, \phi[\vec{h}_{\text{hi}}], l) \quad \phi' = \phi[h_{\text{ho}} \mapsto v_{\text{ho}}]\ \text{, if } h_{\text{ho}} \neq \bot\text{, else } \phi\end{array}}{(\phi', \mathsf{pubres}_{\mathsf{sop}}(l), \tau) \leftarrow \mathit{SimSop}^{\gamma}_{\mathsf{sop}}\,(\phi, \vec{v}_{\text{pi}}, \vec{h}_{\text{hi}}, h_{\text{ho}}, l)}$$

Fig. 9. Simulation of secret operations in γ.

Notice that experiments of Fig. 8 enforce both correctness and security between α and β. For operation output, we effectively ask for π_{output} to be a reshare protocol, with a strong notion of simulator (security) as detailed above. For the remaining operations, we ask for the weaker notion of privacy. The intuition is that, by adding a final resharing step in π_{output}, all shares are reconstructed using fresh randomness, thus cancelling any information that would be revealed by shares produced by intermediate operations.

4.2. Security theorem

Let us refocus on the vertical dimension of Fig. 1. According to Definition 1, we need to exhibit a simulator that, while interacting with the ideal-world semantics, will make it impossible for the environment $\mathcal{Z}$ to distinguish it from an interaction with the real-world semantics.

Simulator. Intuitively, the goal of the simulator $\mathcal{S}$ is to intercept queries made by the adversary $\mathcal{A}$, and construct answers that will trick $\mathcal{A}$ into thinking it is dealing with a distributed program evaluation. In practice, it amounts to constructing a *simulated real semantics*, that shall keep in its internal state a full picture of (a emulation of) the real-world semantics. Looking at rules defining the real-world semantics (Section 2.3), we observe that rule RealSem-stepP, performing a public step on party i, can easily be embedded in the simulator as it can store and manage local configurations for all existing parties. However, the rule RealSem-stepS, that performs evaluation of secret operations, poses a clear challenge: the simulator cannot evaluate secret operations in β, as it does not have access to its internal state. That's where elementary simulators come into play – they will be able to reconstruct β's behaviour based solely on the secret state of corrupt parties and leakage that is made available by the ideal semantics. To that respect, it is convenient to construct a *simulated API library* γ, that keeps in its internal state a secret store containing only shares of corrupted parties, so that it can provide them when needed by elementary simulators $\mathcal{S}_{\mathsf{sop}}$. Fig. 9 presents such a simulated API γ, with $\mathit{SimSop}_{\mathsf{sop}}$ taking the role of $\mathit{EvalSop}_{\mathsf{sop}}$ with an additional *leakage* argument. Note also that the handling of input and output, there is no access to I/O buffers – the required corrupted shares are retrieved from the leakage.

A final ingredient in implementing the simulated real semantics is to control "when" it should interact with the ideal semantics, so that it can retrieve the leakage l when needed by calls to $\mathit{SimSop}^{\gamma}_{\mathsf{sop}}$. Recall that both the ideal and real worlds

are executing the same program. It is then enough to enforce lockstep synchronisation between the ideal-world semantics and some selected party on the emulated semantics.

EasyCrypt formalisation. A better insight of the above description is obtained by looking at its programmatic realisation in our formalisation.

```
module SimulatedRealSem = { ... }
module SimSem(ISem: Sim_Sem_t): Adv_Sem_t = {
  proc stepP(i: int): bool = {
    var b;
    if (i = 1) { ISem.step(); }
    b <@ SimulatedRealSem.stepP(i);
    return b;
  }
  proc stepS(): SideInfo option = {
    var tau;
    tau ← None;
    if (sync(SimulatedRealSem.sigma)) {
      tau <@ ISem.step();
      tau <@ SimulatedRealSem.stepS(oget tau);
    }
    return tau;
  }
}.
module Sim(A: Adv_t, ISem: Sim_Sem_t) = {
  proc init = A(SimSem(ISem)).init
  proc activate = A(SimSem(ISem)).activate
}.
```

Module `SimulatedRealSem` implements the modified version of the real-word semantics. It is similar to `RealSem`, but the `stepS` method has access to an additional leakage argument so that it can replace calls to $\mathit{EvalSop}^{\beta}_{\mathsf{sop}}$ with $\mathit{SimSop}^{\gamma}_{\mathsf{sop}}$. The `SimSem` module responds to adversarial queries based on responses from `SimulatedRealSem`, while resorting to ideal semantics oracle (`ISem`) to obtain the leakage (operator `sync` returns true if all local configurations rest in the same call state). Additionally, module `SimSem` ensures synchronisation if the ideal semantics is with (the emulated) party 1. Finally, module `Sim` uses adversary `A` replacing its oracle with the simulated semantics.

Security theorem. We are now ready to establish our security theorem corresponding to the vertical dimension of Fig. 1.

Theorem 1 *(Single-program secure computation). Under the assumption that the real library β is a secure realisation of the ideal library α, the multiparty evaluation of n copies of a program P under the real-world semantics based on β securely computes P in the ideal-world semantics based on α. Specifically, for arbitrary environment $\mathcal{Z}$ and adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}(A)$ such that*

$$\mathsf{REAL}[\mathcal{Z},\mathcal{A}]](P,\ldots,P) \sim \mathsf{IDEAL}[\mathcal{Z},\mathcal{S}(\mathcal{A})](P).$$

Proof. The proof relies on simulator $\mathcal{S}(A)$ presented above (module `Sim` of the formalisation). It amounts to proving that the adversarial view in the real-world semantics is indistinguishable from its view in the simulated semantics. In EasyCrypt's logic, this is proven by induction on the (secure) transitions of the program (that is the same in both worlds), with an invariant stating that the views observed so far by the adversary are indistinguishable. For single steps, it reduces to two cases: asynchronous single-party steps and synchronous multi-party steps. Because local steps do not entail communication, the simulated semantics does not need to reconstruct any information. Moreover, due to the language determinism and the public nature of all data manipulated by P, local configurations for all parties in the real and simulated semantics are indeed equal, as well as the local configuration of party 1 and the single party of the ideal-world semantics. This ensures that the real and ideal worlds reach synchronisation points (ergo API calls) at the same time. For synchronous steps, we rely on the invariant that the secret store of the simulated semantics (γ) is a projection of corrupted shares from the secret store of real-world semantics (β). Moreover, for each sop, the combination of calls to both α and γ on one side, and calls to β on the other side, reduces to the assumption that β is a secure realisation of α (equivalences of Fig. 8). □

In conclusion, the security of the multiparty program evaluation can be reduced to the security of the underlying API, that is, the security of the collection of protocols that compose it. This approach naturally allows reasoning about the security of unsafe or non-termination programs, and contrasts, e.g., with our previous proof approach for non-interactive programs [20], that relied on composition theorems for individual protocols and imposed safety and correctness restrictions on functionalities.

5. Horizontal dimension: multi-program secure computation

We now turn our attention to the horizontal dimension of Fig. 1, that extends Definition 1 for the case where ideal and real programs may be written in different languages, along with the change in API. This change of languages may occur at two different levels: in the ideal world, by refining the source program (e.g. compiling a high-level specification to a architecture-dependent machine language) before compiling it into a set of multiparty programs; or in the real world, where each party may be, e.g., running in a different architecture and compile its local copy of the source program with a different compiler.

In both scenarios, these horizontal compilers are expected to preserve secure operations of the source programs. Even if the source language provides some high-level abstraction or programming discipline to distinguish secure data processing from non-secure operations, such abstractions are not formally enforced all the way down by traditional compilers, that may ignore such security-aware information. More realistically, even for a domain-specific security-aware compiler, aggressive compiler optimisations or the likelihood of implementation bugs during complex program transformations may introduce security vulnerabilities.

Motivated by safety-critical and high-assurance scenarios, there is nowadays a vast line of programming languages research on constructing *certified compilers*, most notably CompCert [13], which come with a semantic preservation property entailing that the generated target code behaves exactly as prescribed by the source semantics. This section demonstrates how these can be applied to secure compilation for SMC and make the diagram from Fig. 1 commute.

5.1. Notions of certified compilation

Since the behaviour of source and target programs naturally differ in language-specific ways, the classical approach to certified compilation is to reason about source and target semantics that associate (one or more) *observable behaviour* to programs, and establish the semantic preservation property only over observable behaviour. Intuitively, observable behaviour captures the interaction between a program and an external environment (over which it has no control and/or whose behaviour cannot be modelled in the language), including I/O, network communication or system calls. This leaves more margin for the compiler to perform interesting transformations and optimisations. For example, this property is flexible enough to justify the correctness of a traditional dead code elimination transformation, that may remove unnecessary internal behaviour from a program, as long as it does not affect the interactions with the environment.

In our particular SMC scenario, observable behaviour maps directly to security-sensitive interactions with the secure environment for I/O and with the secure API for processing secure operations. As for certified compilation, these secure interactions are not even expressible within the language: I/O involves processing shares, or a probabilistic functionality (e.g., tossing a coin) that cannot be written in the language.

5.1.1. Big-step simulation

Certified compilation approaches typically equip source and target languages with trace-producing big-step semantics for whole programs $P \Downarrow B$, denoting that program P executes with observable behaviour B. Usually [13,21], observable behaviour for whole programs includes termination with a finite trace of observations, divergence with a finite or infinite trace, or *going wrong* for undefined behaviour with a finite trace.

The two classical notions of semantic preservation are to establish forward or backward simulations between source (P_S) and target (P_T) programs. Forward simulation states that all source behaviours shall be preserved by target programs.

Definition 3 *(Forward simulation).* For all B, $P_S \Downarrow B$ implies $P_T \Downarrow B$.

Nevertheless, P_T may have more (possibly undesirable) behaviours than those specified in P_S. Backward simulation conversely states that all target behaviours shall be preserved by source programs.

Definition 4 *(Backward simulation).* For all B, $P_T \Downarrow B$ implies $P_S \Downarrow B$.

In this case, P_S may have more than one possible behaviour, and the compiler is free to choose one of the possible behaviours for the target program.

These definitions differ, therefore, when the semantics of source or target programs are not observably deterministic, i.e., accept more than one possible observation (for the same inputs). We can state this property in a similar way.

Definition 5 *(Observable determinism).* For two B_1 and B_2, if $P \Downarrow B_1$ and $P \Downarrow B_2$, then $B_1 = B_2$.

For observably deterministic programs, these notions collapse and entail the stronger classical notion of bisimulation, where source and target programs must have exactly the same behaviour.

Definition 6 *(Bisimulation).* For all B, $P_S \Downarrow B$ iff $P_T \Downarrow B$.

Realistic certified compilers such as CompCert often relax these properties to hold only for safe (non-wrong) behaviours, or in other words, those considered by the formal semantics of the languages. One typical example is to allow the compiler to eliminate a unused division by zero.

The formal verification approach followed by CompCert is to prove a forward simulation from C to Assembly programs, only for safe behaviours, together with a proof that the Assembly semantics is observably deterministic. This establishes a bisimulation between source C programs and target Assembly programs, for the language subsets where their semantics are formally defined.

5.1.2. Small-step simulation

The operational semantics for source and target languages are classically defined as labelled small-step transitions $S \xrightarrow{B} S'$ between program states S and S', producing a trace of observable behaviour B (with an empty trace ε for local transitions), similarly to our ideal semantics. Whole-program executions and (finite and infinite) traces are constructed in a standard way by executing multiple steps of the small-step semantics and concatenating individual traces B. The notations $S \xrightarrow{B}{}^{+}_{S}{}' $ and $S \xrightarrow{B}{}^{*}_{S}{}'$ respectively denote transitive closure (one or more steps) and reflexive transitive closure (zero or more steps).

The usual way to establish semantic preservation is, therefore, to construct a relation $S_1 \approx S_2$ between states S_1 and S_2 of two languages. Such relation establishes the necessary consistency between states and shall be read as an invariant stating that *so far, the simulation property is preserved*. The proof is then performed by showing that the relation holds for initial states of two programs, and is preserved after performing transitions on both sides.

The simplest (and stronger) property that guarantees semantic preservation is to relate individual transitions.

Definition 7 *(Lock-step simulation).* For two $S_1 \approx S_2$, if $S_1 \xrightarrow{B} S_1'$ and $S_2 \xrightarrow{B} S_2'$, then $S_1' \approx S_2'$.

Lock-step simulation can be easily used to prove the above forward and backward simulation properties for whole programs. Lock-step simulation is also useful for proving properties about the same program (or programs) with the same number of transitions. For instance, the proof of our security Theorem 1 lifts this concept to probabilistic programs, by constructing an equivalence relation between ideal and real views, and showing that steps of the SMC semantics in the ideal and real worlds preserve such relation. Nevertheless, lock-step simulation is admittedly too strong when relating programs written in different languages (as common in compilation) or whose executions require different numbers of transitions (as an optimisation that eliminates redundant transitions).

More useful and relaxed small-step simulation properties thus consider *plus* or *star* simulation, relating sequences of transitions. Nonetheless, although these properties easily imply whole-program semantic preservation for terminating behaviours, additional technical care is needed to ensure whole-program semantic preservation of non-terminating or diverging behaviour.

5.2. Program-based small-step simulation

When mapping the above discussion to the semantical setting presented in Section 2, it should be noted that the call state component of local configurations is actually encoding a calling convention to an external function call. Our basic assumption is that these calls (both arguments and result) are recorded on the trace of the compiler's semantics, and hence preserved throughout compilation. Being the call state a technical artefact of our presentation, it makes sense to identify the state equivalence $\approx$ from a simulation-based proof of behaviour preservation in a certified compiler with a relation on local configurations $\sigma_1 \approx \sigma_2$ (σ_1 and σ_2 respectively on the source and target side), such that:

- $\approx$ is *call-consistent*, that is, $\langle P_1, \rho_1, \zeta_1 \rangle \approx \langle P_2, \rho_2, \zeta_2 \rangle$ implies $\zeta_1 = \zeta_2$;
- $\approx$ satisfies *call-plus backward simulation*: for two related configurations $\sigma_1 \approx \sigma_2$ where $\sigma_2 \rightarrow^{+}_{\mathcal{L}_2} \sigma_2'$ and the call state of σ_2 is non-empty, then there exists a σ_1' such that $\sigma_1 \rightarrow^{+}_{\mathcal{L}_1} \sigma_1'$ and $\sigma_1' \approx \sigma_2'$.

Both properties follow from the trace preservation property, and can readily be exploited in our formalisation. Notice that our languages are deterministic, hence the inverse *call-plus forward simulation* (for $\sigma_1 \approx \sigma_2$ where $\sigma_1 \rightarrow^{+}_{\mathcal{L}_1} \sigma_1'$ and the call state of σ_1 is non-empty, then there exists a σ_2' such that $\sigma_2 \rightarrow^{+}_{\mathcal{L}_2} \sigma_2'$ and $\sigma_1' \approx \sigma_2'$) is implied and simulation is actually a *bisimulation*.[7]

[7] As hinted in Section 2, we could relax the determinism of the languages to *observable determinism*. Our proofs would remain the same and require *call-plus bisimulation*.

$\mathcal{D}(O)$.step():

```
1 : if ¬SimIdealSem_T.sync() then return SimIdealSem_T.step()
2 : info ← ⊥
3 : while (¬sync(SimIdealSem_S.sigma))
4 :     SimIdealSem_S.step(ϵ)
5 :     info ← O.step()
6 : info ← O.step()
7 : if info ≠ ⊥ then
8 :     SimIdealSem_T.step(info)
9 :     SimIdealSem_S.step(info)
10 : return info
```

Fig. 10. Simulator of the target ideal-world semantics ($\mathcal{D}$).

5.3. Ideal certified compilation

Many existing SMC frameworks such as Sharemind [4] or SCALE-MAMBA [5] execute secure computations via a distributed virtual machine: a high-level ideal specification language is compiled to low-level (still ideal) bytecode, and the virtual machine orchestrates the real execution of bytecode among multiple parties. In such approaches, the ideal-world compiler is expected to preserve the behavioural properties of the specification down to the executed bytecode. In this way, we may say it preserves security, since in our framework security-relevant interactions with the system are always reflected on programs' behaviour.

Let $P_S \in \mathcal{L}_S$ and $P_T \in \mathcal{L}_T$ be the source and target programs. In our framework, they would be modelled as two distinct local language semantics, that in turn would be lifted to two ideal-world semantics, both based on the API library α. Security preservation compilation can be framed as the indistinguishability of the associated ideal-world experiments $\mathsf{IDEAL}^{\mathcal{L}_S}$ and $\mathsf{IDEAL}^{\mathcal{L}_T}$, defined as in Section 3.1.

Definition 8 *(Program-based ideal security preservation).* A target program $P_T \in \mathcal{L}_T$ is said to *preserve the ideal security* of a source program $P_S \in \mathcal{L}_S$ if there exists a simulator $\mathcal{D}$ such that, for all adversaries $\mathcal{A}$ and environments $\mathcal{Z}$,

$$\mathsf{IDEAL}^{\mathcal{L}_T}[\mathcal{Z}, \mathcal{A}](P_T) \sim \mathsf{IDEAL}^{\mathcal{L}_S}[\mathcal{Z}, \mathcal{D}(\mathcal{A})](P_S).$$

In order to show the existence of a simulator $\mathcal{D}$ when P_T is obtained from compilation of a source program P_S, we reason pretty much like in Section 3 – we construct a simulated ideal-world semantics, this time for both languages $\mathcal{L}_T$ and $\mathcal{L}_S$, while interacting with the ideal-world semantics for $\mathcal{L}_S$. In a sense, this is a degenerated simulator, as it barely processes the side-information passed around – it only needs to reproduce the interaction pattern for the local execution steps.

We assume that $\mathcal{D}$ has access to the compiler to obtain P_T from P_S. As in proof of Theorem 1, it will embed in its internal state the semantic rules of the ideal-world semantics, but this time it will emulate both the target and source ideal-world semantics (denoted by $\mathsf{SimIdealSem}_T$ and $\mathsf{SimIdealSem}_S$). When defining $\mathsf{SimIdealSem}_T$ and $\mathsf{SimIdealSem}_S$, we encounter the same issue referred when we define SimulatedRealSem (Section 4.2): it doesn't have access to API library α. To circumvent it, we follow the same strategy as before – to define a simulated API library that would provide us enough information to perform the simulation. As we are only interested in the public view of the program evaluation, we can observe that the only relevant information for progressing our emulated semantics are the public results of secret operations. Moreover, we know from the definition of α (Fig. 3) that these public results are computed from the leak, so the very same strategy employed in the construction of simulated API γ (Fig. 9) and the associated SimulatedRealSem module can be applied. Indeed, the only difference is that now we can disregard altogether the secret store, as we are only interested on the public results of secret operations.[8]

Fig. 10 presents the pseudo-code of the simulator $\mathcal{D}$ with access to the source language ideal-world semantics through oracle O. When $\mathcal{D}$ is queried, it inspects the state of $\mathsf{SimIdealSem}_T$ to decide if is performing a local step, in which case it advances it by one step, and returns the result. If $\mathsf{SimIdealSem}_T$ is expecting an API call, it repeatedly advances both $\mathsf{SimIdealSem}_S$ and the ideal-world `step` oracle, until it reaches a non-empty call state (again, by inspecting $\mathsf{SimIdealSem}_S$'s

[8] But we can still use γ, as it provides us with the information needed. The only problem is that it would add unnecessary premises (existence of simulators), and will keep track of information in the secret store that is useless for the case at hand.

$\mathcal{D}(O).\texttt{stepP}(i)$:

```
1 : return SimRealSem^{L_T1 ... L_Tn}.stepP(i)
```

$\mathcal{D}(O).\texttt{stepS}$:

```
1 : τ ← ϵ
2 : if sync(SimRealSem^{L_T1 ... L_Tn}.sigma)
3 :   for i ∈ {1..n}
4 :     res ← T
5 :     while (res = T)
6 :       res ← O.stepP(i)
7 :   τ_S ← O.stepS()
8 :   SimRealSem^{L_T1 ... L_Tn}.stepS(τ_S)
9 : return τ_S
```

Fig. 11. Real world semantics simulator $\mathcal{D}(\mathcal{A})$.

internal state). At the end of the loop, both will be at a secret operation call. It then uses the oracle $O.\texttt{step}()$ to retrieve leakage from the source language ideal-world semantics, and uses it to feed the step methods of both simulated semantics.

Theorem 2 demonstrates how a certified compiler can be used in the compilation of ideal functionalities.

Theorem 2 *(Ideal certified compilation). Let Ω be a compiler bundled with a simulation-based proof of behaviour preservation, and where call/return interactions with the security API are recorded in trace events. For programs P_S and $P_T = \Omega(P_S)$, we have that P_T preserves the ideal security of P_S according to Definition 8.*

Proof. Our proof is a reduction proof using the $\mathcal{D}$ simulator of Fig. 10. Since the two semantics do not run in lock-step, and may perform a different number of steps in different orders, the first part of the proof is to guarantee that the source semantics can catch up to the target semantics when it reaches a call. Since $\mathcal{D}$ only has oracle access to the ideal-world semantics, it animates its own emulation of both programs to inspect when a call state is reached.

Now, assume that two local configurations with $\sigma_1 \approx \sigma_2$. They can either be blocked on the same call state, in which case will receive the same result from the API, or they can be in consistent non-blocked states. By the call-*plus* backward simulation property, if the target semantics reaches a call state, the source semantics will eventually reach the same call state (the while loop terminates), and determinism allows us to conclude call-consistency (both calls are the same). □

By simple composition, any program-based SMC framework can be extended with ideal compilation that, composed with the secure generation of n-party descriptions, achieves a secure low-level evaluation of the program.

Corollary 1. *Let compiler Ω and programs P_S, P_T be as in Theorem 2. If API library β is a secure realisation of α, then, the multiparty evaluation of n copies of a program $P_T, \ldots, P_T$ with API β securely computes P_S with API α.*

5.4. Real certified compilation

Other SMC frameworks such as PICCO [8] transform ideal source programs into party-specific copies where secure operations are replaced by the API code for the respective party, allowing each party to independently compile its local copy with its preferred compiler. In this case, each compiler is expected to preserve the security properties of the source programs for each party.

We can generalise the program-based ideal compilation for multiple parties executing programs from different languages in the real world. As has been noted in Section 2, the real-world semantics could be defined in a setting supporting multiple languages. Let us denote $\mathsf{REAL}^{\mathcal{L}_1 \cdots \mathcal{L}_n}$ as the security experiment as in Section 3.1, but with multi-language support.

Definition 9 *(Program-based real compilation).* A set of target programs $P_{T_1}, \ldots, P_{T_n}$ preserves the real security of a set of source programs $P_{S_1}, \ldots, P_{S_n}$ if there exists a simulator $\mathcal{D}$ such that, for all adversaries $\mathcal{A}$ and environments $\mathcal{Z}$,

$$\mathsf{REAL}^{\mathcal{L}_{T_1} \cdots \mathcal{L}_{T_n}}[\mathcal{Z}, \mathcal{A}](P_{T_1}, \ldots, P_{T_n}) \sim \mathsf{REAL}^{\mathcal{L}_{S_1} \cdots \mathcal{L}_{S_n}}[\mathcal{Z}, \mathcal{D}(\mathcal{A})](P_{S_1}, \ldots, P_{S_n}).$$

Fig. 11 sketches pseudo-code for a concrete simulator for the target real semantics, accessing as an oracle the source real semantics. The approach is similar to the ideal semantics simulator of Fig. 10. It suffices to emulate the target real-world semantics, because the use of the two `step` methods allows for a better control on the evaluation of the source semantics. We construct $\mathsf{SimRealSem}^{\mathcal{L}_{T_1} \cdots \mathcal{L}_{T_n}}$, initialised by compiling source programs $P_{S_1}, \ldots, P_{S_n}$, and again resorting to a simulated API library that simply computes the public result from the leakage (this time, extracted from the communication trace τ).

Theorem 3 establishes the real-world dual of Theorem 2.

Theorem 3 *(Real certified compilation). Let $\Omega_1, \ldots, \Omega_n$ be compilers bundled with simulation-based proofs of behaviour preservation. For programs P_{S_i} and $P_{T_i} = \Omega(P_{S_i})$ $(i \in \{1..n\})$, it holds that programs $P_{T_1}, \ldots P_{T_n}$ preserve the real security of $P_{S_1}, \ldots P_{S_n}$.*

Proof. Our proof is a reduction proof using the $\mathcal{D}(O)$ simulator of Fig. 11. The argument is similar to the ideal case. It relies on emulating the target semantics to keep track of the evaluation of each party. Synchronisation between source and target sides only occurs at API calls (synchronisation points), where all parties on the source side advance (loop of line 4, whose termination follows from backward simulation). Once synchronised, `stepS` is performed on the source and, if successful (that is, I/O does not fail), a further `stepS` is performed on the emulated target semantics to keep it synchronised. □

Finally, by simple composition, any program-based SMC framework can be extended with real compilation.

Corollary 2 *(Real multi-program secure computation). Let $\Omega_1, \ldots, \Omega_n$ be compilers bundled with simulation-based proofs of behaviour preservation. For programs P_S and $P_{T_i} = \Omega(P_S)$ $(i \in \{1..n\})$, it holds that programs $P_{T_1}, \ldots P_{T_n}$ securely compute program P_S with API α. Then, the multiparty evaluation of programs $P_{T_1}, \ldots P_{T_n}$ with API β securely computes P with API α.*

6. Conclusion

In this paper, we have proposed a program-based framework for SMC (Fig. 1), where we decompose the analysis of secure compilation into two orthogonal dimensions: secure orchestration of calls to a static library of low-level protocols (what we call the vertical direction) and transforming the way in which the orchestrations are expressed by leveraging standard certified compilation concepts (the horizontal direction).

Albeit abstract in nature, our framework exposes the fundamental structure and the transformation steps performed by many existing practical SMC frameworks, shedding light into the assumptions made at each level. We hope our work can be used to guide the exploration of new design perspectives for building high-assurance, formally verified practical SMC frameworks, considering arbitrary combinations of certified language-level transformations and secure SMC-level transformations. We illustrate the feasibility of verifying the correctness and security of such frameworks using on our general approach by giving a fully machine-checked formalisation of our results in EasyCrypt.

Although clearer, the road to constructing a concrete SMC framework based on our abstract framework is still challenging. We are currently working on extending the CompCert compiler to support the verified compilation of C programs into popular SMC backends. For simplicity and generality, we have assumed that programs are already partitioned into a local language and secure API operations. In more realistic scenarios, ideal-world programs may be optimised by rearranging which computations are to be run securely or, e.g., compile from a high-level API offering secret variables to a low-level API with a limited number of secret registers and explicit API-level register allocation. Standard functional correctness shall be proven for such additional transformations.

Declaration of competing interest

The authors declare that they have no known conflicts of interest.

Acknowledgements

José Bacelar Almeida was partially funded by the PassCert project, a CMU Portugal Exploratory Project funded by Fundação para a Ciência e Tecnologia (FCT), with reference CMU/TIC/0006/2019.

Appendix A. Notation

Language	
$\mathcal{L}, \mathcal{M}, \mathcal{Z}$	language
$\mathcal{V}$	value domain
$\mathcal{N}$	integer domain
$\mathcal{B} = \{\mathrm{T}, \mathrm{F}\}$	boolean domain
$b \in \mathcal{B}$	boolean value
$\neg b$	boolean negation
P	program
$v \in \mathcal{V}$	value
$h \in \mathcal{V}$	secret value handle
$\emptyset \in \mathcal{V}$	default value
$\sigma = \langle p, \rho, \zeta \rangle$	local configuration
ρ	local environment
$\zeta = \mathsf{Call}_{\mathsf{sop}}(\ldots) \mid \mathsf{Ret}(\ldots)$	local call state

(*continued on next page*)

Language	
$\cdot_\perp$	optional construct
$\perp$	option empty value
$\rightarrow_{\mathcal{L}}$	small-step semantics
$\cdot^\star$	sequence type
$\vec{\cdot}$	sequence
$[]$	empty sequence
$[\cdot, \ldots]$	non-empty sequence
$\xrightarrow{i}$	real-world local semantics
$\stackrel{\tau}{\Longrightarrow}$	real-world distributed semantics
$\Sigma = [\sigma^i]_{i \in \mathcal{P}}$	local party configurations
$\Sigma(i)$	get local party configuration
$\Sigma[i \mapsto \sigma]$	update local party configuration

Secure computation API	
API	secure computation API
$\alpha \in \mathsf{API}$	ideal API
$\beta \in \mathsf{API}$	real API
$\mathcal{P}$	set of parties
$i \in \mathcal{P}$	party
$\bar{\mathcal{V}}$	secret-shared value domain
$\bar{c} \in \bar{\mathcal{V}}$	secret-shared value
share	randomised sharing operation
unshare	unsharing operation
sop = input | output | ...	secure operation
$\mathsf{ar} : \mathsf{sop} \rightarrow \mathcal{N} \times \mathcal{N} \times \mathcal{B} \times \mathcal{B}$	arity of secure operations
$\theta = \langle \phi, \mathcal{I}, \mathcal{O} \rangle$	API state
ϕ	internal API store
$\phi(h)$	get API store value/shares
$\phi[h \mapsto \cdot]$	update API store value
$\mathcal{I}$	API input buffer
$\mathcal{O}$	API output buffer
$(\theta', v_{\mathrm{po}}, \tau) \leftarrow EvalSop^{\mathsf{API}}_{\mathsf{sop}}(\theta, \vec{v}_{\mathrm{pi}}, \vec{h}_{\mathrm{hi}}, h_{\mathrm{ho}})$	evaluation of a secure operation
$(\phi', v_{\mathrm{po}}, \tau) \leftarrow SimSop^{\gamma}_{\mathsf{sop}}(\phi, \vec{v}_{\mathrm{pi}}, \vec{h}_{\mathrm{hi}}, h_{\mathrm{ho}}, l)$	simulation of a secure operation
$\mathcal{T}$	side-information domain
$\tau \in \mathcal{T}$	protocol side-information
$l \in \mathcal{T}$	functionality side-information
$\epsilon \in \mathcal{T}$	empty side-information
$\mathcal{F}_{\mathsf{sop}} : \mathcal{V}^\star \times \mathcal{V}^\star \rightarrow \mathcal{V} \times \mathcal{T}$	functionality of a secure operation
π_{sop}	protocol of a secure operation
$\mathcal{S}_{\mathsf{sop}}$	simulator of a secure operation
$\mathsf{pubres}_{\mathsf{sop}} : \mathcal{T} \rightarrow \mathcal{V}$	leakage public value
$l = \mathsf{leak}^{\pi}_{\mathsf{sop}}(\tau)$	leakage extractor
$\mathcal{C} \subset \mathcal{P}$	set of corrupted parties
$\mathcal{C}(\bar{v})$	shares of corrupted parties

Security model	
$\mathcal{Z}$	environment
$\mathcal{A}$	adversary
$\mathcal{S}, \mathcal{D}$	simulator
$\mathcal{S}(\mathcal{A}), \mathcal{D}(\mathcal{A})$	simulated adversary
$\mathsf{IDEAL}(\mathcal{Z}, \mathcal{A})(P)$	ideal game
$\mathsf{REAL}[\mathcal{Z}, \mathcal{A}](P_1, \ldots, P_n)$	real game
$\mathsf{REAL}^{\mathcal{L}_1 \ldots \mathcal{L}_n}[\mathcal{Z}, \mathcal{A}](P_1, \ldots, P_n)$	multi-language real game
n	number of parties

Certified compilation	
B	observable behaviour
$P \Downarrow B$	big-step semantics
S	language state
ε	empty observable behaviour
$S \xrightarrow{B} S'$	small-step semantics
$S \xrightarrow{B}{}^{+} S'$	one or more steps
$S \xrightarrow{B}{}^{*} S'$	zero or more steps
$S_1 \approx S_2$	relation on states
$\sigma_1 \approx \sigma_2$	relation on local configurations
Ω	compiler

Appendix B. Related work

SMC frameworks. A recent literature review [10] covers 11 SMC compilation frameworks. Conceptually, all such compilers fit into our architecture (Fig. 1) and can be very broadly classified into two major families: those based on garbled circuits (GC) [22], where the application is compiled into the two-party execution of boolean or arithmetic circuit description; and those based on secret sharing [23] (the approach adopted in this paper), where the application is compiled into the multi-party execution of a sequence of SMC protocols for common secure operations. The SMC compilers (including 8 out of the 11 reviewed in [10]) that consider GCs follow the top-right diagonal of Fig. 1—they convert high-level programs into an intermediate circuit description (top), that is passed to a SMC runtime which executes a SMC protocol to compute a result (right). However, our separation between language and secure domain is more amenable to secret sharing approaches, since GC approaches typically express the computation as a single monolithic circuit that is evaluated in one-shot and not by a (programmable) sequence of calls to a lower-level API.

FRESCO [7] is an open-source Java framework allowing secure computation, where functions are described as abstract circuits. The actual gate-by-gate evaluation can be instantiated at runtime by specifying SMC protocols to be used for each gate. Sharemind [4] is a commercial framework for secret sharing SMC with security against passive adversaries, that translates C-like program in a high-level language to a bytecode that is executed by a distributed virtual machine. Individual and composite protocols satisfy the notion of privacy [19] used in our framework. SCALE-MAMBA [5] is a secret sharing SMC framework with security against active adversaries. Python-like SCALE programs are compiled to bytecode in the MAMBA engine, that executes the SPDZ [6] protocol separated into an offline phase, where pre-processing data is generated, and an online phase where parties collaboratively compute SMC operations. PICCO [8] is a SMC compiler that follows the left-bottom diagonal of Fig. 1. It features a source-to-source step that translates security-annotated C programs to party-specific C programs instrumented with calls to a custom secret sharing SMC library with security against passive adversaries (left), such that each party can compile its programs with a regular C compiler and execute it by connecting to a distributed runtime (bottom).

There are few SMC frameworks that provide machine-checked guarantees of the security of the generated protocols. CircGen [14] is a verified compiler translates C programs into boolean GCs, by extending the CompCert C compiler with an additional backend translation from an intermediate register language to a GC. The GC can be evaluated with a, also verified, GC evaluator that guarantees that no information besides the output of the circuit is revealed. Like in our framework, the security guarantees of CircGen rely on CompCert's semantic preservation property, precisely, on the generated GCs having the same I/O behaviour as the original C program. Nonetheless, the GC and secret-sharing security contexts are very different, which is reflected in the kinds of observable behaviour that the compiler is expected to preserve. Since GCs evaluate a whole circuit monolithically and securely, CircGen does not support interactive programs and relies on treating the initial inputs and final outputs of the program, the only interaction points of the GC, as observable behaviour. In contrast, our framework models atomic secure operations as observable behaviour, allowing them to be mixed with non-confidential normal code and relying on an abstract external API to provide their SMC semantics.

Wysteria [15], and its successor Wys$*$ [24], provides a verified toolchain for SMC based on F$*$. Its high-level functional language, similarly to our program-based SC framework, separates non-interactive programs into local computations and secure blocks, where each secure block is compiled to a GC and evaluated by an external GC evaluator. Wysteria's approach fits our vertical dimension of Fig. 1, and formally relates a single-threaded semantics, where a TTP computes the program, to a multi-threaded semantics, where the two parties run the same local code asynchronously but synchronise to execute different code for secure blocks. It ensures language-based information-flow security (one party's execution of the program is independent from the other party's input) by treating secure blocks as idealised black-boxes (that combine secret inputs and produce secret outputs).

The inspiration for this paper comes from previous work [20] that developed a (non-verified) formal framework for passively-secure secret sharing SMC, closely matching the vertical dimension of Fig. 1. The motivation for both works is, however, very different. In [20], the emphasis is on applying language-based information flow to reason about the ideal security of ideal programs written in a C-like language, and showing how (probabilistic) language-based information flow can express the real security of underlying protocols. Moreover, [20] establishes a (vertical) secure compilation result: the underlying protocols, when executed by a distributed semantics of the language, preserve the ideal security of ideal programs, expressed in a single party semantics of the language, and information flow security of protocols implies their cryptographic security. In this work, we construct program-based SMC framework that: i. considers an abstract language, promoting an explicit separation between the language's semantics and the secure computation API; ii. reasons directly about the cryptographic security of the programs. This not only allows us to consider more realistic (horizontal) scenarios where the languages may change, but also permits lifting a few restrictions imposed by the approach of [20]. Namely, we support reactive (and possibly non-deterministic) functionalities, and establish security even for unsafe or non-terminating programs. Also, the language-based protocol security definitions from [20] witness the existence of a simulator for the cryptographically secure protocols used in this work.

Our work is complementary to the verification of SMC protocols in the cryptographic sense. Some of those works focus on the verification that low-level protocols (e.g., components in the static library of the framework) are secure in the specific setting required by an SMC framework. An example is the Sharemind protocol language [25], a domain-specific functional language with a compiler to a passively-secure SMC protocol. Other works focus on formalising cryptography-style security

proofs in interactive theorem provers cryptographic SMC protocols [26–31,14]. We are not verifying the security of individual protocols, but rather the correctness and security of software frameworks that rely on such protocols to build complex secure computations. Nevertheless, we are still in the passive domain, albeit already considering the reactive scenario.

Secure compilation. There is a long line of research on language-based secure compilation [32], i.e., ensuring that compiled target programs preserve the security properties of source programs. The vast majority of these works exploit the same notion of observable behaviour traces used for compiler correctness, and characterise security as a property of (possibly sets of) traces. As a recent concrete example, the work from [33] modifies the CompCert C compiler to capture and preserve a cryptographic constant-time information flow property.

A particular approach that has been gaining more traction lately is to achieve fully abstract compilation [32], i.e., ensuring that (partial) target programs protect source-level security even in the presence of adversarial target contexts (that can be linked against partial programs to construct whole programs). In particular, this requires being able to capture all target-level attacks at the source level. The work from [21] provides an exhaustive characterisation of different kinds of interesting trace properties and related full abstraction enforcement mechanisms. A recent position paper [34] advocates a connection between UC and abstract compilation, in the sense that they both postulate security against arbitrary adversaries. Our results indicate that these dimensions should, in fact, be seen as orthogonal: UC is related to the security of the semantic domains over which secret computations are performed, whereas language-based compilation allows us to move from one language to another, while guaranteeing that the secure semantic domain (an arbitrary context) is used consistently.

References

[1] P. Bogetoft, D.L. Christensen, I. Damgård, M. Geisler, T. Jakobsen, M. Krøigaard, J.D. Nielsen, J.B. Nielsen, K. Nielsen, J. Pagter, et al., Secure multiparty computation goes live, in: International Conference on Financial Cryptography and Data Security, Springer, 2009, pp. 325–343.

[2] L. Kamm, J. Willemson, Secure floating point arithmetic and private satellite collision analysis, Int. J. Comput. Sci. Inf. Secur. 14 (6) (2015) 531–548.

[3] D.W. Archer, D. Bogdanov, Y. Lindell, L. Kamm, K. Nielsen, J.I. Pagter, N.P. Smart, R.N. Wright, From keys to databases—real-world applications of secure multi-party computation, Comput. J. 61 (12) (2018) 1749–1771.

[4] D. Bogdanov, S. Laur, J. Willemson, Sharemind: a framework for fast privacy-preserving computations, in: European Symposium on Research in Computer Security, Springer, 2008, pp. 192–206.

[5] A. Aly, M. Keller, E. Orsini, D. Rotaru, P. Scholl, N.P. Smart, T. Wood, Scale–mamba v1. 3: Documentation, Tech. rep., Technical Report, 2019.

[6] I. Damgård, V. Pastro, N. Smart, S. Zakarias, Multiparty computation from somewhat homomorphic encryption, in: Annual Cryptology Conference, Springer, 2012, pp. 643–662.

[7] I. Damgård, K. Damgård, K. Nielsen, P.S. Nordholt, T. Toft, Confidential benchmarking based on multiparty computation, in: International Conference on Financial Cryptography and Data Security, Springer, 2016, pp. 169–187.

[8] Y. Zhang, M. Blanton, G. Almashaqbeh, Implementing support for pointers to private data in a general-purpose secure multi-party compiler, ACM Trans. Priv. Secur. (TOPS) 21 (2) (2018) 6.

[9] K. Bab, Jiff: Javascript implementation of federated functionality, https://github.com/multiparty/jiff, 2019.

[10] M. Hastings, B. Hemenway, D. Noble, S. Zdancewic, SoK: General Purpose Compilers for Secure Multi-Party Computation, in: 2019 IEEE Symposium on Security and Privacy (S&P), IEEE, 2019, pp. 1220–1237.

[11] I. Damgård, J.B. Nielsen, Universally composable efficient multiparty computation from threshold homomorphic encryption, in: Annual International Cryptology Conference, Springer, 2003, pp. 247–264.

[12] R. Canetti, Universally composable security: a new paradigm for cryptographic protocols, in: Proceedings 42nd IEEE Symposium on Foundations of Computer Science, IEEE, 2001, pp. 136–145.

[13] X. Leroy, Formal certification of a compiler back-end or: programming a compiler with a proof assistant, in: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2006, pp. 42–54.

[14] J.B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, B. Grégoire, V. Laporte, V. Pereira, A fast and verified software stack for secure function evaluation, in: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, 2017, pp. 1989–2006.

[15] A. Rastogi, M.A. Hammer, M. Hicks, Wysteria: a programming language for generic, mixed-mode multiparty computations, in: 2014 IEEE Symposium on Security and Privacy, IEEE, 2014, pp. 655–670.

[16] A. Rastogi, P. Mardziel, M. Hicks, M.A. Hammer, Knowledge inference for optimizing secure multi-party computation, in: Proceedings of the Eighth ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, 2013, pp. 3–14.

[17] N. Büscher, D. Demmler, S. Katzenbeisser, D. Kretzmer, T. Schneider, Hycc: compilation of hybrid protocols for practical secure computation, in: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, 2018, pp. 847–861.

[18] K.R. O'Neill, M.R. Clarkson, S. Chong, Information-flow security for interactive programs, in: 19th IEEE Computer Security Foundations Workshop (CSFW'06), IEEE, 2006, 12 pp.

[19] D. Bogdanov, P. Laud, S. Laur, P. Pullonen, From input private to universally composable secure multi-party computation primitives, in: 2014 IEEE 27th Computer Security Foundations Symposium, IEEE, 2014, pp. 184–198.

[20] J.B. Almeida, M. Barbosa, G. Barthe, H. Pacheco, V. Pereira, B. Portela, Enforcing ideal-world leakage bounds in real-world secret sharing mpc frameworks, in: 2018 IEEE 31st Computer Security Foundations Symposium (CSF), IEEE, 2018, pp. 132–146.

[21] C. Abate, R. Blanco, D. Garg, C. Hritcu, M. Patrignani, J. Thibault, Journey beyond full abstraction: exploring robust property preservation for secure compilation, in: 2019 IEEE 32nd Computer Security Foundations Symposium (CSF), IEEE, 2019, pp. 256–25615.

[22] A.C.-C. Yao, How to generate and exchange secrets, in: 27th Annual Symposium on Foundations of Computer Science (sfcs 1986), IEEE, 1986, pp. 162–167.

[23] A. Shamir, How to share a secret, Commun. ACM 22 (11) (1979) 612–613.

[24] A. Rastogi, N. Swamy, M. Hicks, Wys*: a dsl for verified secure multi-party computations, in: International Conference on Principles of Security and Trust, Springer, 2019, pp. 99–122.

[25] P. Laud, J. Randmets, A domain-specific language for low-level secure multiparty computation protocols, in: Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security, ACM, 2015, pp. 1492–1503.

[26] R. Canetti, A. Stoughton, M. Varia, Easyuc: using easycrypt to mechanize proofs of universally composable security, in: 32nd IEEE Computer Security Foundations Symposium (CSF 2019), 2019.

[27] K. Eldefrawy, V. Pereira, A high-assurance evaluator for machine-checked secure multiparty computation, in: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, 2019, pp. 851–868.
[28] H. Haagh, A. Karbyshev, S. Oechsner, B. Spitters, P.-Y. Strub, Computer-aided proofs for multiparty computation with active security, in: 2018 IEEE 31st Computer Security Foundations Symposium (CSF), IEEE, 2018, pp. 119–131.
[29] F. Kerschbaum, Automatically optimizing secure computation, in: Proceedings of the 18th ACM Conference on Computer and Communications Security, ACM, 2011, pp. 703–714.
[30] J. Bacelar Almeida, M. Barbosa, E. Bangerter, G. Barthe, S. Krenn, S. Zanella Béguelin, Full proof cryptography: verifiable compilation of efficient zero-knowledge protocols, in: Proceedings of the 2012 ACM Conference on Computer and Communications Security, ACM, 2012, pp. 488–500.
[31] M. Pettai, P. Laud, Automatic proofs of privacy of secure multi-party computation protocols against active adversaries, in: 2015 IEEE 28th Computer Security Foundations Symposium, IEEE, 2015, pp. 75–89.
[32] M. Patrignani, A. Ahmed, D. Clarke, Formal approaches to secure compilation: a survey of fully abstract compilation and related work, ACM Comput. Surv. (CSUR) 51 (6) (2019) 1–36.
[33] G. Barthe, S. Blazy, B. Grégoire, R. Hutin, V. Laporte, D. Pichardie, A. Trieu, Formal verification of a constant-time preserving c compiler, Proc. ACM Program. Lang. 4 (POPL) (2019) 1–30.
[34] M. Patrignani, R.S. Wahby, R. Künneman, Universal composability is secure compilation, arXiv preprint, arXiv:1910.08634, 2019.