Universal Composability is Robust Compilation

MARCO PATRIGNANI*, University of Trento, Italy
ROBERT KÜNNEMANN, CISPA Helmholz Center for Information Security, Germany
RIAD S. WAHBY†, Carnegie Mellon University, USA
ETHAN CECCHETTI, University of Wisconsin–Madison, USA

This paper discusses the relationship between two frameworks: universal composability (UC) and robust compilation (*RC*). In cryptography, UC is a framework for the specification and analysis of cryptographic protocols with a strong compositionality guarantee: UC protocols remain secure even when composed with other protocols. In programming language security, *RC* is a novel framework for determining secure compilation by proving whether compiled programs are as secure as their source-level counterparts no matter what target-level code they interact with. Presently, these disciplines are studied in isolation, though we argue that there is a deep connection between them and exploring this connection will benefit both research fields.

This paper formally proves the connection between UC and *RC* and then it explores the benefits of this connection (focussing on perfect, rather than computational UC). For this, this paper first identifies which conditions must programming languages fulfil in order to possibly attain UC-like composition. Then, it proves UC of both an existing and a new commitment protocol as a corollary of the related compilers attaining *RC*. Finally, it mechanises these proofs in Deepsec, obtaining symbolic guarantees that the protocol is indeed UC.

Our connection lays the groundwork towards a better and deeper understanding of both UC and *RC*, and the benefits we showcase from this connection provide evidence of scalable mechanised proofs for UC.

In order to differentiate various sub-parts of the UC *and RC frameworks, we use* syntax highlighting *to a degree that colourblind and black&white readers can benefit from [78].*

UC *talks about ideal functionalities (typeset in a* verbatim, emerald *font) and protocol implementations (typeset in a* sans-serif, orange *font). RC deals with source languages (typeset in an* italics, blue *font) and target ones (typeset in a* **bold**, **red** *font). Elements that are common to each framework are typeset in a* black, sans-serif *font for* UC *and in a black, italicised font for RC to avoid repetition.*

For a better experience, please view this paper in colour.

CCS Concepts: • **Security and privacy** → **Cryptography**; • **Theory of computation** → **Program semantics**;

Additional Key Words and Phrases: secure compilation, programming language semantics, universal composability, cryptography, composition, proof techniques, backtranslation

ACM Reference Format:
Marco Patrignani, Robert Künnemann, Riad S. Wahby, and Ethan Cecchetti. 2022. Universal Composability is Robust Compilation . *ACM Trans. Program. Lang. Syst.* 1, 1, Article 1 (January 2022), 64 pages. https://doi.org/10.1145/3436809

*Part of this work was conducted while at MPI-SWS, at Cispa, and at Stanford University.
†Part of this work was conducted while at Stanford University.

Authors' addresses: Marco Patrignani, disi, University of Trento, Trento, Italy, mp@cs.stanford.edu; Robert Künnemann, CISPA Helmholz Center for Information Security, Saarbrücken, Germany, robert.kuennemann@cispa.de; Riad S. Wahby, Carnegie Mellon University, Pittsburgh, PA, USA, riad@cmu.edu; Ethan Cecchetti, University of Wisconsin–Madison, Wisconsin, USA, cecchetti@wisc.edu.

0164-0925/2022/1-ART1
https://doi.org/10.1145/3436809

1 INTRODUCTION

In cryptography, universal composability (UC) is a framework for the specification and analysis of cryptographic protocols with a key guarantee about compositionality [24, 39, 40, 60, 69, 75]. If a cryptographic protocol is proven UC, that protocol behaves like some high-level, secure-by-construction *ideal functionality* no matter what the protocol interacts with. As such, if that protocol is used within a larger protocol, in order to reason about the latter, we can replace the former protocol with its ideal functionality and just reason about the rest. In other words, UCprotocols are secure *even when composed* with other protocols.

In programming language security, robust compilation (*RC*) [12, 14] is a framework for studying secure compilation as the *robust* preservation of classes of hyperproperties [45]. That is, for each class $\mathcal{H}$ of hyperproperties (e.g., safety, hypersafety, subset-closed hyperproperties etc.), *RC* provides a criterion stating that if attained by a compiler, then the compiler preserves class $\mathcal{H}$ from the source programs it inputs to the target ones it produces. Moreover, this preservation is done robustly, i.e., *no matter* what code is linked against the source and target programs.

These two frameworks (Section 2) belong to different research fields and seem to deal with different notions, however we demonstrate that they are deeply connected. Both frameworks are concerned with abstract notions which are generally *deemed* secure, and more concrete notions whose security must be *proven against arbitrary attackers*. In UC, the abstract notions are called ideal functionalities (F) while concrete ones are called protocols (π). In *RC*, the abstract notions are called source programs (P), for the source language of the compiler $[\![\cdot]\!]$, while concrete ones are target programs ($\mathbf{P}$), for the target language of the compiler. Moreover, proving that a protocol UC-realises an idea functionality ($\pi \vdash_{\text{UC}} \mathsf{F}$), or that a compiler attains a *RC* criterion for class $\mathcal{H}$ ($\vdash [\![\cdot]\!] : \mathcal{H}$), is done by showing that any attack at the concrete level (protocol or target program) can be simulated at the abstract level (functionality or source program).

This paper is the first to witness (and formally prove in Isabelle) the connection between these two frameworks (Section 3). Specifically, we connect UC to the *RC* criterion that corresponds to *R*obust *H*yperproperties *P*reservation (dubbed *RHP*). We detail all assumptions typically made by the two frameworks and clarify which additional (simple) assumptions need to hold for this connection to hold. Effectively, we demonstrate that proving $\pi \vdash_{\text{UC}} \mathsf{F}$ is equivalent to: code F as a program P, code π as a program $\mathbf{P}$, and prove that $\vdash [\![\cdot]\!] : \mathit{RHP}$ for the compiler that translates P into $\mathbf{P}$.

After presenting this connection, this paper discusses the benefits that both frameworks gain from it. Admittedly, the benefits we identify, and reap, benefit the UC world more than the *RC* one, in fact the paper focusses on the possibility of obtaining rigorous, scalable, mechanised proofs of UC from *RC* proofs. For this, we note that UC results are stated in terms of the semantics of Interactive Turing Machines (ITMs), while *RC* ones are stated in terms of arbitrary source and target languages. Thus, we first identify which conditions arbitrary programming languages must fulfil in order to be used to attain UC proofs from *RC* ones (Section 4). These conditions formalise the behaviour of the linking operators of such languages, and they impose some constraint on the languages' semantics. In addition, these conditions let us derive common UC corollaries that simplify the mechanisation of *RC* proofs.

We then present one language that we prove to fulfil these conditions: the *reactive, interactive λ-calculus* (RILC, Section 5). RILC extends the interactive λ-calculus of Liao et al. [73] with a module system and a trace semantics, which are needed in order to carry out *RC* proofs. The module system is needed to clearly demarcate the boundary between protocol and attacker (resp. functionality and simulator). The trace semantics describes the behaviour of a protocol linked with an attacker from the perspective of an external observer (i.e., the environment perspective, in UC terms), as

commonly done in reactive systems formalisations [18]. Together, these additions yield a language where we can express all the concepts required by UC, but still perform proofs in a *RC* style, leveraging the formal language semantics.

Next, we demonstrate how the semantics-based proof techniques used for *RC* proofs can be leveraged to obtain rigorous, scalable UC proofs (Section 6). We discuss structural properties and automation of both UC and *RC* proofs in the literature. This includes formalisations of UC that are mechanized and allow for rigorous proofs, but could so far not exploit the connection to *RC*. For this we take a protocol for one-time commitments π_{comm} [41, 73] and prove that it UC-realises some ideal functionality $\mathsf{F}_{\mathsf{comm}}$. We prove this both with static and with dynamic corruption models; to the best of our knowledge the former is a known result [41, 73] but the latter is novel. By relying on our connection, we code the ideal functionality and the protocols as programs (P_{comm} and $\mathbf{P}_{\mathbf{comm}}$ respectively) in RILC. We prove that the compiler $[\![\cdot]\!]$, which translates P_{comm} to $\mathbf{P}_{\mathbf{comm}}$ attains *RHP*, so from our connection, π_{comm} UC-realises $\mathsf{F}_{\mathsf{comm}}$. We carry out this proof for both the static and the dynamic corruption models, and both proofs rely on simplifications of a trace-based backtranslation, a semantics-based proof technique often used in *RC* proofs [11, 12, 14, 51, 79, 82, 83, 85]. While similar proof approaches for UC have been used in the literature (e.g., in computer-aided crypytography), our connection is the first to formally bridge the gap between the UC setting using ITMs and that in which other results are stated. The proofs that we show imply *RC* proof techniques, but they nevertheless follow a pattern that is already known in the literature.

Finally, we mechanise both these proofs in DEEPSEC [44], a fully automated protocol verification tool for privacy properties. We translate P_{comm} and $\mathbf{P}_{\mathbf{comm}}$ in the languages of DEEPSEC (which is very close to RILC) and use the tool to prove that the programs are trace equivalent, so in turn, the compiler from the former to the latter is *RHP*.

The connection between UC and programming languages semantics had already been conjectured, but the programming-languages counterpart had been (wrongly) identified [58] to be fully-abstract compilation [1, 80] (interestingly, the first formal criterion for secure compilation). We discuss this conjecture and other elements of our connection in Section 8, before presenting related work (Section 9) and concluding (Section 10).

Concretely, this paper makes the following contributions:

- it proves in Isabelle that (under some simple conditions), UC coincides with *RHP*, the hyperproperty preservation instance of *RC*;
- in doing so, it disproves an existing conjecture [58] claiming that UC is a different secure compilation criterion called fully-abstract compilation;
- it provides the conditions for arbitrary languages to be used for our connection;
- it formalises RILC, the reactive, interactive λ-calculus and proves it fulfils the aforementioned conditions (together with other useful properties);
- it proves a one-time commitment protocol to UC-realise some ideal functionality under both static and dynamic corruption models by proving the compiler from functionality to protocol is *RHP*;
- it mechanises these proofs in DEEPSEC, obtaining symbolic UC guarantees.

For the sake of simplicity, we delegate much of the auxiliary formalism to the technical report [86], where the interested reader can find full language formalisation, auxiliary lemmas and proofs. Mechanised proofs for the main results (in Isabelle) and for the commitment protocol (in DEEPSEC) can be found at our project page:

https://uc-is-sc.github.io/

2 BACKGROUND: UC AND *RC*

This section presents the UC (Section 2.1) and *RC* (Section 2.2) frameworks and the details that we rely on to define our connection between them.

A Note on Abstraction. There exist several UC frameworks and a lot of programming languages that can fit the *RC* framework. In the following, the details of both frameworks (e.g., the choice of interactive Turing machines or the language semantics) are deliberately abstract and handled in an axiomatic manner. Such an abstract treatment lets us derive the connection in the most general sense, but it may leave the knowledgeable reader concerned about some technical details. These technical details are handled and discussed later on, when we show how concrete UC frameworks and what real languages fulfil these axioms and provide a concrete instance of our connection.

2.1 Universal Composability

In UC [37, 39] the involved parties form a collection of *Interactive Turing Machines* (ITM), that is, Turing Machines that operate concurrently and that have shared tapes between each other. Crucially: in a collection of ITMs, only a single machine operates at a time, so computation happens under sequential consistency.

ITMs. The framework of UC talks about entities in two distinct worlds: the *real* and the *ideal* ones. The entities of each world comprise exactly the same roles: one encodes the protocol of interest, one models the adversary and one that models an environment providing inputs and observing behaviour. In the real world, these entities are: the protocol π, the adversary A and the environment Z. In the ideal world, these entities are: the ideal functionality F, the simulator S and the (same) environment Z. In either world, the three entities form a *network* that executes according to the semantics of ITMs.

ITMs Semantics. ITMs have a deterministic semantics whose source of randomness comes from a specific tape called the random tape that any machine in the network can read from. The final outcome of the interaction of a network of ITMs (as perceived by the environment) is (wlog) a single bit 0/1 [37, §1.1]. Thus, the semantics of a network of ITMs (e.g., π, A, Z) can be expressed as a random variable $\textsc{Exec}\,(\pi, \mathsf{A}, \mathsf{Z})$ whose image is a single bit.

Oftentimes, e.g., in UC proof arguments, it is useful to describe the sequence of messages exchanged by the entities in the network. We call this sequence of messages a *trace*, to uniform the lexicon with the one of *RC* (described in Section 2.2 below). Thus, the trace of a network of ITMs (e.g., π, A, Z) can be expressed as a random variable $\textsc{ExecT}\,(\pi, \mathsf{A}, \mathsf{Z})$ whose image is a sequence of messages.

Definition 1 (ITMs Semantics in UC)**.**

$$\textsc{Exec}\,(\cdot,\cdot,\cdot) : 0/1 \qquad\qquad \textsc{ExecT}\,(\cdot,\cdot,\cdot) : \text{ trace of messages}$$

The precise definition of ExecT and Exec differs from framework to framework, so in Section 3.2.1 we capture their essence with a set of axioms that abstract from the framework-specific (gory) details. We verified that these axioms hold for the UC-like frameworks [36, 37, 73], as we report in Section 8.1.

The semantics of ITMs plays a role in the security argument of UC, which is about the real world emulating the ideal one. This means indistinguishability of executions i.e., of the random variable Exec when run on a real and on an ideal network.

UC-*Emulation and Indistinguishability.* UC distinguishes different forms of emulation, the two most important ones being *perfect* and *computational* emulation [61, 62]. In this work we focus on perfect emulation, and we leave discussing computational emulation for future work. In perfect emulation, the real and ideal worlds are considered indistinguishable ($\approx$) if their semantics are equally distributed. In general, for two random variables X, Y in the same observation space E (bits in our case), this means that $X \approx Y \iff \forall e \in E. \Pr[X = e] = \Pr[Y = e]$, where $\Pr[c]$ indicates the probability of condition c.

UC *Security.* We now have all the elements to define the UC security argument (Definition 2). In UC, a protocol π is secure (denoted with $\pi \vdash^{=}_{\text{UC}} \mathsf{F}$) if it refines an ideal functionality F that is secure by construction (aka, the protocol π UC-emulates the functionality F). UC ensures that π is as secure as F by requiring that any attack on π can be rewritten into an attack on F. This is expressed via simulation: π emulates F if for any adversary A, there is a simulated adversary S such that, for any environment Z, the networks $(\mathsf{Z}, \pi, \mathsf{A})$ and $(\mathsf{Z}, \mathsf{F}, \mathsf{S})$ are indistinguishable.

Definition 2 (UC Perfect Emulation)**.**

$$\pi \vdash^{=}_{\text{UC}} \mathsf{F} \stackrel{\text{def}}{=} \forall \mathsf{A}.\ \exists \mathsf{S}.\ \forall \mathsf{Z}.\ \textsc{Exec}\,(\mathsf{Z}, \mathsf{A}, \pi) \approx \textsc{Exec}\,(\mathsf{Z}, \mathsf{S}, \mathsf{F})$$

2.2 Robust Compilation

Robust compilation criteria [12, 14] are defined for a generic notion of a source language $\mathcal{S}$ and a target language $\mathcal{T}$ so long as they provide a few elements that we now describe.

Formal Languages, Traces and Semantics. In *RC*, languages must have a notion of partial programs P (or, components) and of attackers to the programs (or, program contexts) A. The two can be linked together $A \bowtie P$[1] and the result is a another program W that we call *complete*. Intuitively, a complete program is one that has no external dependencies. Notice however that a complete program can still be linked with other programs that use the code of the former program.[2] For example, a program may be the implementation of an encryption library and it can be linked with an attacker that interacts with said library, yielding a complete program. This complete program can then be linked within a larger program, say modelling a web browser, that uses the encryption library. While this setup with complete programs is not canonical in the *RC* setting (which is built on programming language semantics theory), we justify it later, when reasoning about composition of programs (and attackers, and complete programs) in Section 4.1.1.

Languages must provide a representation of their behaviour through traces $\overline{\tau}$ which are an infinite sequence of security-relevant events. The trace model considered by Abate et al. [14] is that of infinite traces performed against an interface *common* to both source and target languages (e.g., syscalls).[3] The trace model also considers prefixes ($\overline{\mu}$), i.e., finite traces.

To talk meaningfully about cryptographic operations, we expect the language semantics to take a randomness parameter in input. This, in turn, changes the trace model and traces become a pair of a sequence of actions ($\overline{\tau}$, as before) together with the probability for that sequence to happen (ρ). Formally, we indicate a trace and a prefix as follows:

$$\text{Interaction Trace } \overline{\tau} ::= \text{ infinite sequence of messages} \qquad \text{Trace } \overline{t} ::= (\overline{\tau}, \rho)$$

[1]Linking is often denoted as $A\,[P]$, we use a different notation here to drive a cleaner visual connection with UC.
[2]We refrain from using the word 'whole' since in programming language semantics it has a slightly different connotation than the one of this paper. Essentially, canonical whole programs cannot be extended anymore with other code, while our complete programs can (as we show in Section 4).
[3]Having a common trace model across language is a common assumption in compiler works [14, 72], though existing work has shown how to lift this assumption [12, 83]. We keep this assumption in our development for the sake of simplicity.

$$\text{Interaction Prefix } \overline{\mu} ::= \text{ finite sequence of messages} \qquad \text{Prefix } \overline{m} ::= (\overline{\mu}, \rho)$$

Technically, the trace model may be something different than what we present here, so long as it is something that agrees with the UC trace model.

To connect traces and prefixes, we rely on the prefixing operation, denoted with $\leq$. Let $\overline{\mu} \leq \overline{\tau}$ be the canonical prefixing operation between prefix $\overline{\mu}$ and trace $\overline{\tau}$, i.e., the finite sequence of messages of $\overline{\mu}$ is contained in the infinite sequence of messages of $\overline{\tau}$. Since traces (and prefixes) are augmented with probabilities, we lift the prefixing operator $\leq$ to our trace model by requiring the trace part to be a prefix and the probability part to be the same.

$$(\overline{\mu}, \rho) \leq (\overline{\tau}, \rho') \stackrel{\text{def}}{=} \overline{\mu} \leq \overline{\tau} \text{ and } \rho = \rho'$$

We leave the semantics of our languages abstract, and indicate the reduction relation of the semantics with $\rightsquigarrow$. However, there must be a way to indicate the behaviour of programs in terms of the set of traces generated by a program according to the semantics of the language:

$$Behav(W) \stackrel{\text{def}}{=} \left\{ \overline{t} \;\middle|\; W \rightsquigarrow \overline{t} \right\}$$

and indicate that two programs W_1 and W_2 have the same behaviours as follows:

$$W_1 \eqsim W_2 \stackrel{\text{def}}{=} Behav(W_1) = Behav(W_2)$$

Since in this paper we assume a common trace model, shared between all languages, note that $\eqsim$ can also be used to compare whether programs in different languages have the same behaviour or not. The $\eqsim$ relation is reflexive, symmetric and transitive, i.e., it is an equivalence. As an example, below are the traces that describe the behaviour of a program that returns a single random bit once queried:

$$\{(\text{Query?} \cdot \text{Reply } 0!, 1/2), (\text{Query?} \cdot \text{Reply } 1!, 1/2)\}$$

The semantics of the languages that we consider (which includes the semantics of most programming languages, including ITMs) is supposed to have a simple property that, for lack of a better word, we call 'being ok'. Intuitively, for any program W, a semantics is *ok* if, given that it can produce all prefixes of a trace, then it produces the trace too, with the same probability.

Definition 3 (Ok Semantics)**.**

$$\vdash \rightsquigarrow : ok \stackrel{\text{def}}{=} \forall W, \overline{t}.\ \text{if } (\ \text{if } \forall \overline{m}.\ \overline{m} \leq \overline{t} \text{ then } W \rightsquigarrow \overline{m}) \text{ then } W \rightsquigarrow \overline{t}$$

Hyperproperties. Hyperproperties [45] are a formal representation of predicates on programs, i.e., they are predicates on sets of traces. They capture many security-relevant properties including conventional safety and liveness (i.e., predicates on traces), as well as properties like non-interference (i.e., predicates on sets of traces). The class of hyperproperties we focus on (for now) is *arbitrary hyperproperties* (H).

Compilers. The last element we need to introduce for *RC* are compilers. Given a source language S and a target language T, the compiler that translates S components P into T ones is denoted with $[\![S]\!]^{S}_{\mathrm{T}}$. When the languages of the compiler are not relevant, we simply write $[\![\cdot]\!]$.

We take compilers to be partial functions, and indicate a compiler to be undefined for certain inputs i as $[\![i]\!] = \bot$. The criteria we define (and use) in this paper only hold for those inputs for which a compiler is defined.

RHP. Below we present *RHP* (Definition 4), a criterion that is fulfilled by compilers that preserve subset-closed hyperproperties robustly. Intuitively, a compiler attains *RHP* if the behaviour ($\mathbf{Behav}(\cdot)$) of the compiled program ($[\![P]\!]$) linked ($\bowtie$) with an arbitrary target attacker ($\mathbf{A}$) is the same as the behaviour ($Behav(\cdot)$) of the source program (P) linked ($\bowtie$) with a source attacker (A). If we unfold the definition of behaviour, we obtain that a compiler attains *RHP* if and only if any prefix ($\overline{m}$) produced ($\rightsquigarrow$) by the compiled program linked with an arbitrary target attacker can be replicated ($\rightsquigarrow$) by the source program linked with a source attacker.

Definition 4 (Robust HP Preservation)**.**

$$[\![\cdot]\!] \vdash RHP \stackrel{\text{def}}{=}, \mathbf{A}.\ \exists A.\ \mathbf{A} \bowtie [\![P]\!] \eqsim A \bowtie P$$

$$\text{or equivalently: } \forall P, \mathbf{A}.\ \exists A.\ \mathbf{Behav}(\mathbf{A} \bowtie [\![P]\!]) = Behav(A \bowtie P)$$

$$\text{or equivalently: } \forall P, \mathbf{A}.\ \exists A.\ \forall \overline{m}.\ \mathbf{A} \bowtie [\![P]\!] \rightsquigarrow \overline{m} \text{ iff } A \bowtie P \rightsquigarrow \overline{m}$$

By looking at the statement of *RHP* it is not clear that amounts to the preservation of all H. Such a result has been provided by Abate et al. [14], as they prove that *RHP* is equivalent to the statement of Definition 5 that more clearly amounts to the preservation of all H. We refer the interested reader to the work of Abate et al. [14] for a proof of the equivalence of the two definitions.

Definition 5 (Robust HP-Preserving Compiler)**.**

$$[\![\cdot]\!] \vdash RHP \stackrel{\text{def}}{=} \forall H.\ \forall P.\ \text{ if } (\forall A.\ Behav(A \bowtie P) \in H) \text{ then } (\forall \mathbf{A}.\ \mathbf{Behav}(\mathbf{A} \bowtie [\![P]\!]) \in H)$$

Having defined the two frameworks of interest, we now describe the connection between them.

3 CONNECTING THE FRAMEWORKS

This section first discusses our connection visually and informally (Section 3.1) before formally proving it (Section 3.2).

3.1 Informal Connection

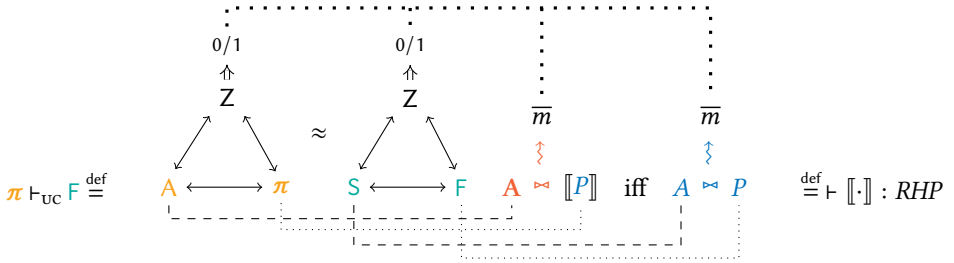

Fig. 1. Connecting UC and *RHP*, visually.

Figure 1 presents a visual depiction of our connection. There, (almost) each element in a framework has a corresponding counterpart in the other. Protocols π correspond to compiled programs $[\![P]\!]$, ideal functionalities F correspond to source programs P and the environment Z corresponds to the prefix $\overline{m}$. Concrete adversaries A correspont to target attackers $\mathbf{A}$ and existentially-quantified simulators S correspond to existentially-quantified source attackers A. However, the connection between certain elements is non-trivial, and we discuss these cases below.

In the following, we use the word *system* to mean one of the four pairs at the bottom of each triangle in the figure (i.e., what in *RC* we called a complete program). As such, the system is always

composed of an adversary (or an attacker, or a simulator) and another entity: the protocol, the functionality, the source program or its compiled counterpart.

The Role of Environments. The connection between the environment Z and the prefix $\overline{m}$ is most apparent when thinking about their role, which is observing the system and providing input to it.

A difference between environments and prefixes is their security role—at least at the intuitive level. In UC, the environment is a *benign* entity, so it is not supposed to provide messages that deviate from the protocol and its observation is assumed to be fair. In *RC*, a prefix is just an objective indication that some reduction has happened in the system, so a prefix cannot be benign or malicious.

Fortunately, from the formal perspective, the environment in UC is an ITM that has no way to be restricted to just benign behaviour. Instead, since the environment is universally quantified, it can behave both in a benign and in a malicious way. For this reason, formally, the (malicious) adversary role can be subsumed by the environment, and in UC it is therefore possible to replace the adversary with a dummy forwarder (this is the dummy attacker theorem, as we discuss in Section 4.2).

On the other hand, there are languages in which prefixes are not an objective indication of a reduction happening, but they are generated by a computing entity that the system is reacting to: these are called *reactive languages*. Our connection works with both reactive and non-reactive languages, and we discuss how using reactive ones simplifies certain results (such as porting the dummy attacker theorem from UC to *RC*) in Section 8.3. However, we focus on reactive languages since non-reactive ones can have infinite traces, which leads to weaker results than UC [76]

Traces VS Single Bit. A seeming discrepancy in the connection is the single-bit differentiation output of UC contrasted with a full prefix in *RC*. We argue this is merely a cosmetic difference: in UC all parties exchange messages – whose concatenation forms indeed a prefix – and the bit is a function of those messages (as captured by Axiom 2). It is therefore just for convenience (and a technicality due to dealing with probability distributions) that the environment outputs a single bit in order to tell whether it is interacting with the real or ideal world.

What are Compilers in UC*?* In the UC framework there is no element that corresponds to compilers. Instead, the translation of high-level `functionalities` into protocols is human made. As such, the typical UC translation risks being imprecise and error prone, and it cannot be automated for large functionalities, or between functionalities with recurring parts.

On the other hand, in *RC* the compilers we consider are not what one expects: they do not traverse the program in input and translate its subparts. Instead, they perform a syntactic check on the whole source program: if it is a specific one (i.e., the ideal functionality), then the related protocol is produced by the compiler. In turn, this means that the compilers we consider are not total functions on their input, but partial ones, defined only on those source programs that are ideal functionalities. Notation-wise, we identify such partial functions as $[\![P]\!] \rightharpoonup \mathbf{P}$.

Note that in this work we still do not provide a compiler for the systematic translation of functionalities into protocols. We envision such a compiler exists, for a protocol-specification language, and leave a discussion of the kind of compiler we consider for now for Section 3.2.4.

3.2 Formally Proving the Connection

The proof strategy we adopt to formally prove the connection is depicted in Figure 2. Our main result is the dashed co-implication on top, which represents the equivalence of UC and *RC* (Theorem 3 in Section 3.2.4). We break down that equivalence into two other equivalences, the sloped, full co-implications on the bottom (Theorem 1 and Theorem 2).

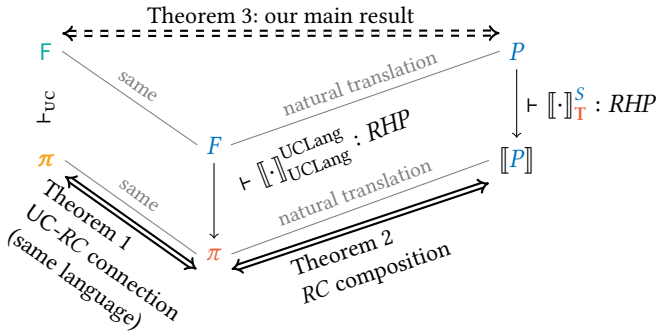

Fig. 2. Visual depiction of the proof of Theorem 3.

First we must establish the formal semantics of UC. Despite there being a number of flavours of UC, we define an abstract language UCLang that captures the essence of all these flavours. We define the semantics of UCLang via a set of axioms (Section 3.2.1); later (Section 8.1), we show that many UC flavours respect these axioms.

With UCLang, we can prove the connection between UC and *RC* provided that we are using UCLang in the *RC* statements (Theorem 1 in Section 3.2.2). That is, the compiler we consider must have the same source and target language: UCLang, and we indicate this compiler as $[\![\cdot]\!]^{\text{UCLang}}_{\text{UCLang}}$. Since the semantics of ITMs (in UC) and of UCLang (in *RC*) is the same, in the figure we indicate that F (resp. π) and F (resp. π) are the 'same'.

Once this connection is proven, we lift the restriction on the languages of the compiler and consider any compiler between an arbitrary source and an arbitrary target language (Section 3.2.3). We assume a "natural translation", i.e., a way to translate programs between S and UCLang and between T and UCLang that preserves the same behaviour. We argue that such natural translations already exist, since real-world protocols are not written in ITMs but in programming languages. So finally, by composing these natural translations with $[\![\cdot]\!]^{\text{UCLang}}_{\text{UCLang}}$ we obtain $[\![\cdot]\!]^{S}_{\mathrm{T}}$. By relying on existing results about the composition of translations [66] we obtain that $[\![\cdot]\!]^{S}_{\mathrm{T}}$ is also *RHP* (Theorem 2).

3.2.1 Axioms for UC *Semantics.* The first axiom relates the UC semantics of ITMs to the one of UCLang, a programming language semantics in the style of *RC*, and it defines what we mean when an adversary and a program produce a trace (Axiom 1).[4] In this axiom (as well as in subsequent ones) we assume a function $z(\cdot) : \overline{m} \to \mathrm{Z}$ that assigns each possible trace a *canonical environment*. A canonical environment for $\overline{m}$ produces exactly this prefix and then halts the execution with the final bit 1. Such a canonical environment is a known concept in the programming languages research literature, it is the environment obtained from the backtranslation of a trace [11, 14, 65, 79, 81, 83]. We will discuss this in more detail in Section 6.1.2.

Axiom 1 (UC and UCLang Semantics)**.**

$$A \bowtie P \rightsquigarrow \overline{m} \text{ iff } \Pr[\textsc{ExecT}(z(\overline{m}), \mathrm{A}, \mathrm{P}) = \overline{\mu}] = \rho > 0 \qquad \text{where } \overline{m} = (\overline{\mu}, \rho)$$

(Formalized in `UC.relation_uc`.)

[4]This axiom is typeset in black since it is valid for connecting source and ideal functionality semantics as well as trace and real protocol semantics. Technically both pairs talk about the same semantics since source and target languages here are UCLang and real and ideal world talk about ITMs.

Next, prefixes must hold enough information to extract whether the environment has decided to output a final bit b or not ($\bot$). In UC [37], the environment terminates the trace with the final bit, hence any prefix must have it. Abstracting away from the encoding, we assume an extraction function $\beta : \overline{m} \to \{0, 1, \bot\}$.

Axiom 2 (Finite Traces Contain the Final Bit)**.**

$$\Pr[\textsc{Exec}\,(\mathsf{Z}, \mathsf{A}, \pi) = b] = \sum_{\beta(\overline{m})=b} \Pr[\textsc{ExecT}\,(\mathsf{Z}, \mathsf{A}, \pi) = \overline{\mu}]$$

(Formalized in `UC.prefix_iff_trace_is_prefix`.)

To be sure that the canonical environment does not introduce undesired behaviour, we require that it is correct (Axiom 3). We define correctness to mean $z\,(\overline{m})$ can produce $\overline{m}$ if any environment can, and that it distinguishes $\overline{m}$ from all other executions by only outputting the final bit $\beta(\overline{m})$ only when the environment sees exactly the prefix it is looking for. The side conditions required for this axiom ensure that the canonical environment exists for a non-probabilistic environment Z that can produce the prefix at hand.

Axiom 3 (Canonical Environment Correctness)**.** For all non-probabilistic Z that produce some prefix $\overline{m}$, i.e., $\exists \mathsf{A}_e, \pi_e.\, \Pr[\textsc{ExecT}\,(\mathsf{Z}, \mathsf{A}_e, \pi_e) = \overline{m}] > 0$, we have

$$\begin{aligned}\Pr[\textsc{ExecT}\,(\mathsf{Z}, \mathsf{A}, \pi) = \overline{m}] &= \Pr[\textsc{ExecT}\,(z\,(\overline{m})\,, \mathsf{A}, \pi) = \overline{m}] \\ &= \Pr[\textsc{Exec}\,(z\,(\overline{m})\,, \mathsf{A}, \pi) = \beta(\overline{m})]\end{aligned}$$

(Formalized in `UC.construct_canonical_env` and `UC.construct_canonical_env2`.)

The construction of this canonical environment is straightforward in most UC frameworks: it matches each incoming input against the expected input in the prefix and hard-codes its output. Thus, intuitively, the canonical environment reduces the environment to its input/output behaviour.

In the axiom we assume Z is non-probabilistic, meaning that the probability of that prefix depends only on the randomness used in A and π. This is not a real limitation, since non-probabilistic environments are complete (Axiom 4).

Axiom 4 (Non-Probabilistic Environments are Complete)**.**

$$\text{if } \textsc{Exec}\,(\mathsf{Z}, \mathsf{A}, \pi) \not\approx \textsc{Exec}\,(\mathsf{Z}, \mathsf{S}, \mathsf{F}) \text{ then } \exists \text{ non-probabilistic } \mathsf{Z}_n.\, \textsc{Exec}\,(\mathsf{Z}_n, \mathsf{A}, \pi) \not\approx \textsc{Exec}\,(\mathsf{Z}_n, \mathsf{S}, \mathsf{F})$$

(Formalized in `UC.nonprobabilistic_envs_are_complete`.)

For UC, Canetti [37, p. 47] states this is the case. Intuitively, if there is a distinguishing environment, then there is at least one set of random choices that we can condition the environment's randomness on such that the environment still succeeds in distinguishing both systems. Since the environment is quantified after the adversary and protocol (Definition 2), we can hard-code these choices into the environment, making it deterministic.

3.2.2 Connecting UC *and RHP Between UCLang.* With the semantics of UC defined via the axioms, we can prove a simplified version of our connection. If a protocol UC-realises a functionality, then the UCLang-compiler from that functionality to that protocol is *RHP* (Theorem 1).

Theorem 1 (UC and *RHP* Coincide for UCLang)**.**

$$[\![P]\!]^{\text{UCLang}}_{\text{UCLang}} \vdash^{=}_{\text{UC}} P \text{ iff } \vdash [\![\cdot]\!]^{\text{UCLang}}_{\text{UCLang}} : \mathit{RHP}$$

Proof. (Proven in Isabelle/HOL: Theorem `RHPtoUC` and Theorem `UCtoRHP`.)

In both directions, we proceed by contradiction, we only discuss the *RHP* $\Rightarrow$ UC one since the structure of the two directions is the same. At the intuitive level, the proof relies on the axioms of Section 3.2.1 in order to switch between the ExecT representation of ITMs semantics (of the UC framework) to the $\rightsquigarrow$ representation of the same semantics (of the *RC* framework).

RHP $\Rightarrow$ UC By contradiction, we assume the negation of UC (Definition 2), so program P is not emulated by its compiled counterpart $[\![P]\!]$.

This gives us an adversary A such that for any simulator S, there exists an environment Z that can distinguish real and ideal interactions (negation of Definition 2, point 1).

Thus, without loss of generality, we have a prefix $\overline{m}$ whose generating execution is more probable in the real world than in the ideal one (Axiom 2).

This execution can also be generated by feeding $\overline{m}$ to the canonical environment (Axiom 3), so we eliminate the environment in favour of the canonical one.

We now use Axiom 1 twice, on the real and on the ideal world, in order to translate point 1 into *RC*. Thus, we obtain the *RC* counterparts of the UC elements: $\mathbf{A} = \mathsf{A}$ and $A = \mathsf{S}$ such that this holds:

$$\mathbf{A} \bowtie [\![P]\!] \rightsquigarrow \overline{m}$$

but at the same this does not hold:

$$A \bowtie P \rightsquigarrow \overline{m}$$

This is in direct contrast with the *RHP* assumption (Definition 4): contradiction. □

3.2.3 Connecting UC *and RHP Between Any Language.* UC and *RHP* coincide for programs written in UCLang, but what about other languages? To answer this, we need to take a protocol in UCLang and essentially translate it into any other language without changing its meaning: we call this a *natural translation*.

We argue this natural translations are already done in practice, at an abstract level, and in the following, we formalise the meaning of natural translations. In fact, while UC proofs are carried out between ITMs, real protocol implementations are done in real programming languages. Thus, cryptographers already know how to naturally translate between UCLang and any other programming language.

Two programs P and $\mathbf{P}$ are the natural translation of each other if they have the same behaviour robustly.

Definition 6 (Program Natural Translation)**.**

$$P_1 \lesssim P_2 \stackrel{\text{def}}{=} \forall A_1.\, \exists A_2.\, Behav(A_1 \bowtie P_1) = Behav(A_2 \bowtie P_2)$$

$$P \sim \mathbf{P} \stackrel{\text{def}}{=} P \lesssim \mathbf{P} \text{ and } \mathbf{P} \lesssim P$$

The notion of natural translation can then be lifted to languages by ensuring that there are natural translation for all of their programs.[5]

Definition 7 (Language Natural Translation)**.**

$$L_1 \lesssim L_2 \stackrel{\text{def}}{=} \forall P_1 \in L_1.\, \exists P_2 \in L_2.\, P_1 \sim P_2$$

$$L_1 \sim L_2 \stackrel{\text{def}}{=} L_1 \lesssim L_2 \text{ and } L_2 \lesssim L_1$$

[5]So, the natural translation of two languages can witnessed by the existence of a galois insertion between components of each language whose abstraction and concretisation functions are essentially two *RHP* compilers between the languages. Moreover, this insertion must induce equivalence between the behaviours of the insertion-related programs. Since it is not the focus of this paper, we leave exploring the interesting ramifications of this fact for future work.

The next notion we need is that of language restriction w.r.t. a compiler input or output. Specifically, given a language L and a compiler $[\![\cdot]\!]^S_T$, $L|^I_{[\![\cdot]\!]^S_T}$ indicates the language obtained by considering only the programs in the input of $[\![\cdot]\!]^S_T$, i.e., those programs for which the partial compiler is defined $[\![P]\!]^S_T \rightharpoonup \mathbf{P}$. Dually, $L|^O_{[\![\cdot]\!]^S_T}$ indicates the language obtained by considering only the programs in the output of $[\![\cdot]\!]^S_T$.

The final element of technical machinery we need is a way to compare what we call "robust" language expressiveness. After all, the results we are setting up span different languages: the source and target languages of the compiler and UCLang. Therefore, it is unsurprising that we need to compare language expressiveness to enforce that all languages can represent the same programs. Formally, we say that UCLang can express all the behaviour of a compiler $[\![\cdot]\!]^S_T$ if UCLang has a natural translation with the language the compiler takes in input, and with the language the compiler produces as output.

Definition 8 (UCLang Related Expressiveness)**.**

$$\text{UCLang} \vDash [\![\cdot]\!]^S_T \stackrel{\text{def}}{=} S|^I_{[\![\cdot]\!]^S_T} \lesssim \text{UCLang and } T|^O_{[\![\cdot]\!]^S_T} \lesssim \text{UCLang}$$

Finally, we need a way to relate *RHP* compilers between UCLang and between arbitrary languages S and T. If a program and its compilation are natural translations of their UCLang counterparts, then the compiler mapping the program to its compilation is *RHP* if and only if the compiler mapping their UCLang counterparts is also *RHP*.

Theorem 2 (*RHP* for Arbitrary Languages)**.**

$$\begin{aligned} &\text{if } P \sim P_1 \text{ and } \mathbf{P} \sim P_2 \text{ and } [\![P]\!]^S_T \rightharpoonup \mathbf{P} \text{ and } [\![P_1]\!]^{\text{UCLang}}_{\text{UCLang}} \rightharpoonup P_2 \\ &\text{then } \vdash [\![\cdot]\!]^{\text{UCLang}}_{\text{UCLang}} : \mathit{RHP} \text{ iff } \vdash [\![\cdot]\!]^S_T : \mathit{RHP} \end{aligned}$$

Proof. (Proven in Isabelle/HOL: Theorem `implementUC`.) Follows from Theorem 1 (UC and *RHP* Coincide for UCLang) and from the fact that when programs have natural translations as per Definition 6, they produce the same behaviour. □

3.2.4 Our Main Result, Formally. Joining all these results together yields the formal proof of our main result: UC and *RHP* coincide. We first provide a specialised version of our result (Theorem 3) before showing the generalisation (Theorem 4) and wrapping up with a discussion of said results.

Both versions rely on program natural translations in order to bridge the gap between a program (resp. a compiled program) and a functionality (resp. a protocol). The specialised version uses what may seem like a degenerate compiler (in the programming languages sense). Alas, we believe this 'degeneration' may offer insights into protocol language design, which we discuss right after the theorem. The general version, instead, relies on Definition 8 to draw the more general result about *RHP* and UC.

Theorem 3 (UC and *RHP* Coincide)**.**

$$\begin{aligned} &\text{if } P \sim \mathbf{F} \text{ and } \pi \sim \mathbf{P} \text{ and } [\![\cdot]\!]^S_T \stackrel{\text{def}}{=} [\![P]\!]^S_T \rightharpoonup \mathbf{P} \\ &\text{then } \vdash [\![\cdot]\!]^S_T : \mathit{RHP} \text{ iff } \pi \vdash_{\text{UC}} \mathsf{F} \end{aligned}$$

Proof. (Proven in Isabelle/HOL: Theorem `UC_impl_SingletonRC` and Theorem `RC_impl_UC`.) The co-implication of this theorem is split into the two-coimplications depicted in Figure 2. From left to right, we obtain the result from Theorem 2 (*RHP* for Arbitrary Languages) and Theorem 1 (UC and *RHP* Coincide for UCLang). From right to left, we proceed by contradiction. Given an unsimulatable

UC attacker against π, we use the natural translation $P \sim \mathsf{F}$ and $\pi \sim \mathsf{P}$ to construct a counterexample against the assumption that $\vdash [\![\cdot]\!]^S_{\mathrm{T}} : \mathit{RHP}$. □

A Seemingly-Degenerate Compiler. The compiler considered in Theorem 3 is not defined inductively on the grammar of source programs, as often done in programming languages work [19, 64, 72, 83]. Instead, it works as an input-output match: it translates those input programs for which it is defined, into the associated output programs. Ideally, we see such a compiler to have an input-output pair for all known UC results. Then, since protocols are devised as the combination of smaller protocols, we believe the kind of compiler we consider in Theorem 3 can be used to compile larger protocols by composing the compilation of the smaller ones. In essence, this kind of compiler should consider a special source language that is used to express protocols and their composition; since the details of such a language escape the scope of this paper, they are left for future work.

Our results, however, do not just work for such seemingly-degenerate compilers, as demonstrated by Theorem 4. Indeed, the UC-*RC* connection we present works for any compiler that is *RHP*, provided that the compiler translates the program related to the ideal functionality to the program related to the protocol.

Theorem 4 (UC and *RHP* Coincide, Generalised)**.**

$$\text{if UCLang} \vDash [\![\cdot]\!]^S_{\mathrm{T}}$$
$$\text{then } \vdash [\![\cdot]\!]^S_{\mathrm{T}} : \mathit{RHP} \text{ iff } \left(\forall P, \mathsf{P}, \mathsf{F}, \pi.\ \text{if } [\![P]\!]^S_{\mathrm{T}} \rightharpoonup \mathsf{P} \text{ and } P \sim \mathsf{F} \text{ and } [\![P]\!]^S_{\mathrm{T}} \sim \pi \text{ then } \pi \vdash_{\mathrm{UC}} \mathsf{F}\right)$$

Proof. (Proven in Isabelle/HOL: Theorem `UC_impl_RC` and Theorem `RC_impl_UC`.)
Half of the proof ($\Leftarrow$) is identical to Theorem 3 and does not rely on the language expressiveness assumption. The other half uses very similar logic to the proof of Theorem 3. □

Discussion of the Results. A way to summarise our result is thus:

A security proof between ITMs is a UC proof,
a security proof between arbitrary languages is a *RHP* one.

However, oftentimes (mainly for simplicity) we are interested in setting up the connection for the same source and target languages, which we now refer to as just S (i.e., $\mathrm{T} = S$). For example, this is what we do in Section 6. This specialisation requires S and UCLang to have natural translations (as per Definition 7), so the easiest candidates for S are those that are shown to behave like ITMs. Since ILC has been proven to model ITMs faithfully, that is why we will choose it and extend it in order to showcase an instance of our connection.

Overall, we believe the connection between either S or T and UCLang is not interesting: cryptographers have been modelling real-world code as ITMs for decades. Thus, we believe the most important interpretation of Theorem 3 is that one should really focus on the S to T translation and its security proof.

At this point, however, one may wonder whether our connection is as precise as it gets, or whether there are other *RC* notions (or programming languages research notions, as conjectured by Hicks [58]) that connect with UC. We will prove that our connection is the most precise one later on, in Section 8.2.

Using our Connection. We now have a good intuition that the connection holds, and of what it connects. What are we to make of this connection then, and how can we reap its benefits?

In this paper we reap the benefits for UC, attaining rigorous mechanised UC proofs from *RC* ones. We leave investigating the benefits for *RC* coming from the UC connection for future work.

In a nutshell, the first step now is to instantiate abstract languages S and T. The second step is devising a compiler from some ideal functionality written in S to some concrete protocol written in T, and prove that compiler is *RHP*.

In this paper, we set S and T to be the same language: RILC (Section 5). This is the simplest solution that lets us showcase the value of our connection by proving UC of a commitment protocol as a *RHP* proof (Section 6). Building on this language also lets us mechanise this proof (Section 7).

A different approach would be to define a specification language (for S) and model the language in which some protocol is implemented (i.e., T) and prove *RHP* at the level of the compiler from S to T. While mechanisation is more complex in this case – we conjecture such proofs could be mechanised in Coq[6] – its applicability to real-world protocols is clearer. We leave exploring this avenue of results for future work.

The procedure we describe above is not novel, however, as some work in computer-aided cryptography use a similar approach in their UC proofs [27, 56]. However, no existing work provides a formal proof that allows them to bridge the gap between their language and the world of UC. Thus we believe our result provides a formal justification for the proof approach of these works.

Before we showcase our connection, however, we need to unravel a few more assumptions on the semantics of programming languages. In fact, for now we have imposed very few constraints on the semantics of programming languages involved in the connection. Thus, one may wonder whether, in practice, the connection can be instantiated with realistic (formal) languages. This is what we investigate next in Section 4.

4 GENERALISING UC COROLLARIES IN *RC*

In this section, we import two important corollaries from UC that are essential to its use for composing secure protocols: the composition theorem and the dummy attacker one. By proving these corollaries in the *RC* setting, we identify additional conditions for languages that can be used with our connection. Thus this section first proves the composition theorem in the *RC* setting (Section 4.1) and formulates the dummy attacker theorem (Section 4.2), providing us with concrete requirements for secure composition in programming languages.

4.1 The Composition Theorem in *RC*

As discussed in Section 2.1, UC results always talk about the same language: ITMs. As such, the different types of composition (e.g., *protocol composition* between two protocols and *protocol execution* composing adversary, protocol and environment to an executable system [37]) are always between ITMs. Conversely, *RC* results talk about different languages, so when one tries to derive a composition result, one has to deal with composition of programs in different languages. Thus, porting the composition theorem in *RC* lets us identify additional conditions that the composition of programs in different languages must fulfil.

We state these conditions as a set of axioms that regulate the behaviour of *RC* composition in order to be able to port a key UC result in *RC*. Specifically, the UC framework comes with a key free theorem called the *Composition Theorem* (Theorem 5), meaning that any protocol that UC-realises a functionality also enjoys Theorem 5. Intuitively, the composition theorem lets one replace a sub-protocol that is part of a larger protocol with its ideal functionality, which is typically much simpler to reason about. Thus, the composition theorem is the basis for the modular analysis of protocols.

[6]We remark that mechanising proofs of *RC* (and more in general of secure compilation) is not an easy task, and at the time of writing, very few such examples exist [11, 50].

Theorem 5 contains the formal definition of the composition theorem. There, and in the following, we indicate a larger protocol π_{large} composed with a sub-protocol π_{small} (which could also be an ideal functionality) as $\pi_{large}{}^{\pi_{small}}$.

Theorem 5 (Composition Theorem in UC [37])**.**

$$\text{if } \pi_s \vdash_{UC} \mathsf{F}_s \text{ then } \pi_|{}^{\pi_s} \vdash_{UC} \pi_|{}^{\mathsf{F}_s}$$

The theorem states that if a protocol π_s UC-realises a functionality F_s and π_s is used within a larger protocol $\pi_|$, then protocol $\pi_|$ with π_s UC-realises $\pi_|$ with F_s in place of π_{small}.

Why Theorem 5 is a corollary of UC-realisation can be explained by reasoning about the role of the environment. Ideally, the environment represents all possible larger protocols $\pi_|$, so UC-emulation guarantees that no $\pi_|$ can distinguish between its sub-part being π_s or F_s.

The key notion for this result is the composition operator, so we first discuss this operator in UC and in *RC* (Section 4.1.1). Then, we present the set of axioms that regulate the behaviour of composition (Section 4.1.2). As we demonstrate later, providing a concrete instantiation of the *RC* composition is fairly simple, and there exists plenty of such instances in the real world; moreover, equivalently easy is showing that these instances respect the axioms. Finally, we state (and prove) the analogous of Theorem 5 in *RC* (Section 4.1.3).

4.1.1 Composition Operators.

UC *Composition.* Depending on the UC framework, composition is expressed as a program transformation [37, 60], where two protocols are inlined into a wrapper/sandbox ITM, or as a rebinding of input and output channels [36]. In both cases, the composition of $\pi_|$ with π_s into a single protocol $\pi_|{}^{\pi_s}$ restricts the communication interfaces of both protocols. For example, the sub-protocol π_s cannot communicate directly with the environment, but any such communication is routed through $\pi_|$. Additionally, π_s and $\pi_|$ have different interfaces to the adversary, that is, there is an adversary for π_s and one for $\pi_|$. Finally, adversaries for π_s and $\pi_|$ do not communicate directly, but only via the environment.

RC Compostition. When defining *RC* composition we rely on canonical programming language semantics notions and do not model the communication restriction mentioned above for UC. We do this for the sake of simplicity and generality, and in order to reuse much of the existing semantics theory developed for *RC*. We leave investigating the semantics of more intricate composition mechanisms for future work. Essentially, given the visibility qualifiers of a function defined in a program (e.g., `public`, `private`, etc), composition does not change those qualifiers. In UC, composition can change these qualifiers, and composing $\pi_|{}^{\pi_s}$ can turn `public` interfaces of π_s into `private` ones.

Recall from Section 2.2 that our languages have a notion of programs P and of attackers A that can be linked together ($A \bowtie P$) into a complete program. Since the way we define for *RC* can talk about multiple languages, we define multiple composition operators in *RC*, each differing in its signature, as we present below.

linking: $\bowtie$ is the already-presented linking. It models intra-language composition and its signature is $A \to P \to W$. In UC, linking corresponds to the combination of a protocol with an adversary.

program FFI: $\circledast^{S}_{T}$ models cross-language composition of programs, so its signature is $P \to \mathbf{P} \to P$. As a notational convention, cross-language composition symbols are annotated with languages: the top-most is the language of the first argument and of the result, while the bottom-most is the second argument's.

Intuitively, program FFI composition yields a S program made of a $\mathbf{T}$ and an S sub-part (resp, $\mathbf{P}$ and P), that can communicate with each other. When real-world languages let programs of different languages S and $\mathbf{T}$ interoperate with each other, they rely on an actual foreign function interface between S and $\mathbf{T}$. As such, any concrete implementation of this operator must include one such foreign function interface that marshals and unmarshals (i.e., converts) values of one language into the other and vice-versa. In language semantics, this can be modelled as a multilanguage system [20, 74, 87]. We need not use this model in our instantiation of the composition operators (later in this paper, in Sections 5 and 6), because for simplicity, we resort to using a single language for S and $\mathbf{T}$.

This composition operator also exists in UC, where it is dubbed the protocol composition operator, and it has the same signature. In fact, program FFI composition combines two programs into one program while protocol composition combines two protocols into one protocol. Alternatively, certain UC results are formulated in *hybrid* models [38]. There, both functionality and protocol are composed with a functionality that models a setup assumption. For example, if π emulates F in the $\mathsf{F}_{\mathsf{CRS}}$-hybrid model, then π contains the common-reference string functionality to realize F. The composition operator presented here is the one allowing such containment to happen.

attacker FFI: $\oplus^{S}_{\mathbf{T}}$ models a different cross-language parallel composition, since its signature is $A \to \mathbf{A} \to A$.

Although many works let attackers and programs have the same structure [4, 37, 95], this is not always the case [14, 21, 22, 55, 83]. For this reason, we need a composition operator for attackers that may differ from the program one. This operator is not always given a name in UC, but appears as a construction in the composition proof (e.g., [37, Fig. 8]).

complete FFI: $\circledcirc^{S}_{\mathbf{T}}$ models a cross-language composition for complete programs, so its signature is $W \to \mathbf{W} \to W$.

In order to link any syntactic category of our abstract languages, program and attacker FFI are not sufficient. Therefore we define the complete FFI, too. It defines the composition of systems (pairs of protocol plus adversaries), where the former becomes the environment to the latter.

In UC, the environment is an ITM that provides inputs to both the adversary and protocol. It 'represents whatever is external to the current protocol execution. This includes other protocol executions and their adversaries, human users, etc.' [37]. UC's environment (this ITM) thus serves as a proxy for the higher-level system and the *protocol execution*, composing environment and a pair of adversary and protocol, can be seen as a system composition.

4.1.2 Conditions for RC Composition. To understand what can be a valid or an invalid instance of these composition operators, we define axioms that regulate their behaviour. These axioms are eventually used to derive *RHP*, and thus they are stated in terms of behavioural equivalence ($\eqsim$ from Section 2.2).[7]

Axiom 5 requires that any attacker communicating with a program composed of two sub-programs can be decomposed into two attackers, each in the language of the respective program. The reason we need the two programs is to know into what kind of languages to split the attacker. Had we only a single program, we would not know the language of the second attacker.

[7]We conjecture that proving this connection for computational UC may result in a connection with a different *RC* criterion that may not be *RHP*. As such, we believe the following axioms would need to replace $\eqsim$ with whatever is appropriate in that connection. For the sake of generality, our Isabelle formalisation is stated in terms of an arbitrary transitive relation.

Axiom 5 (Attacker Decomposition). $\forall A, P, \mathbf{P}.\ \exists A', \mathbf{A}'.$

$$A \bowtie \left(P \circledast_{\mathbf{T}}^{S} \mathbf{P}\right) \eqsim \left(A' \mathbin{⦶}_{\mathbf{T}}^{S} \mathbf{A}'\right) \bowtie \left(P \circledast_{\mathbf{T}}^{S} \mathbf{P}\right)$$

(Formalized in `composition.par_decomp`.)

Axiom 6 states that linking an attacker and a program that are themselves composed using their FFI compositions is equivalent to linking each attacker and program and then using complete FFI composition for the result.

Axiom 6 (FFI Recombination).

$$\left(A \mathbin{⦶}_{\mathbf{T}}^{S} \mathbf{A}\right) \bowtie \left(P \circledast_{\mathbf{T}}^{S} \mathbf{P}\right) \eqsim (A \bowtie P) \mathbin{◎}_{\mathbf{T}}^{S} (\mathbf{A} \bowtie \mathbf{P})$$

(Formalized in `composition.inter_comp` and `composition.inter_decomp`.)

Finally, we formalise the relation between the behaviour of a program and the composition of this program with a new one. Essentially, we require that any two programs that have the same behaviour can be FFI-composed with *the same* program, and still have the same behaviour.

Axiom 7 (Constant Addition).

$$\text{if } P \eqsim \mathbf{P} \text{ then } P \mathbin{◎}_{S}^{O} P \eqsim P \mathbin{◎}_{\mathbf{T}}^{O} \mathbf{P}$$

(Formalized in `composition.const_elim`.)

Technically, Axiom 7 can be proven as a co-implication instead of as an implication, since our *RC* composition operators do not change the visibility qualifiers within programs and attackers. Since the implication is the required direction for the general composition theorem, we state the weaker axiom.

4.1.3 Deriving the Composition Theorem. We can now state the *RC* analogue to Theorem 5 as Theorem 6. Here, in the conclusion, we identify the simulator (or, source-level attacker) with S in order to differentiate it from the adversary (or target-level attacker) A. Apart from this cosmetic change, the theorem states that if a compiler is *RHP* when linking its target and source with a target attacker and a source simulator, then it is still *RHP* when linking against an external program (P) and then against an external attacker A and simulator S.

Theorem 6 (Composition in *RC*).

$$\text{if } \forall \mathbf{A}.\exists A.\ \mathbf{A} \triangleright\!\bowtie [\![P]\!] \eqsim A \triangleright\!\bowtie P \text{ then } \forall A.\exists S.\ A \triangleright\!\bowtie P \circledast_{\mathbf{T}}^{O} [\![P]\!] \eqsim S \triangleright\!\bowtie P \circledast_{S}^{O} P$$

Proof. (Proven in Isabelle/HOL: Theorem `composition`.) Figure 3 provides a visual depiction of this proof and how each of the presented axioms contribute to proving it (via transitivity of $\eqsim$). Below, we indicate assumption $\mathbf{A} \triangleright\!\bowtie [\![P]\!] \eqsim A \triangleright\!\bowtie P$ with *HP*.

□

The proof to this theorem is remarkably compact (ten lines of Isabelle/HOL) which makes it easier to comprehend and separate the framework-specific insights (which are compartmentalised in the operators and their axioms) from the structure of the proof. In Section 5.1, we will demonstrate that instantiating these composition operators and proving these axioms for a formal language is easy. Moreover, it simplifies the proof of the composition theorem, which is usually considered the main result of any work presenting a UC-like framework.

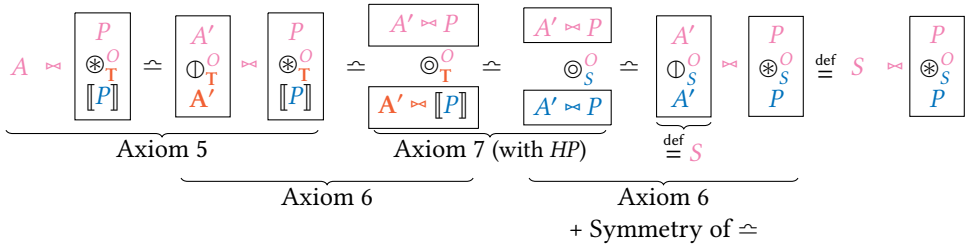

Fig. 3. Visual depiction of the proof of Theorem 6.

4.2 The Dummy Attacker in *RC*

The second key UC result we want in *RC* is the dummy adversary theorem (Theorem 7). Intuitively, Theorem 7 states that instead of considering all possible adversaries, one should consider only a specific one, the *dummy adversary*. This dummy adversary only forwards messages from the environment to the protocol (which perceives them as coming from the adversary), and relays messages from the protocol to the adversary to the environment. The common UC frameworks define the dummy adversary explicitly, e.g., as a stateless ITM that implements a 'transparent channel' [37, 60] or as set of channels between environment and protocol [36]. In the context of *RC*, we cannot explicitly define the dummy adversary, as the concrete formulation is dependent on the programming language. Instead, we will axiomatically define how it interacts with the other part of the system.

Definition 9 (Dummy Adversary (informal))**.** In UC, the dummy adversary ($\mathsf{A_d}$) is a proxy that relays all messages from the environment to the protocol and vice versa.

With the dummy adversary, we can state its theorem:

Theorem 7 (Dummy Adversary Theorem in UC [37])**.**

$$\forall \mathsf{A}.\ \exists \mathsf{S}.\ \forall \mathsf{Z}.\ \textsc{Exec}\,(\mathsf{Z}, \mathsf{A}, \pi) \approx \textsc{Exec}\,(\mathsf{Z}, \mathsf{S}, \mathsf{F}) \text{ iff } \exists \mathsf{S}.\ \forall \mathsf{Z}.\ \textsc{Exec}\,(\mathsf{Z}, \mathsf{A_d}, \pi) \approx \textsc{Exec}\,(\mathsf{Z}, \mathsf{S}, \mathsf{F})$$

From a formal perspective it is simple to understand why this theorem holds, the $\Rightarrow$ direction of the co-implication being trivial and the $\Leftarrow$ direction being the interesting one. In fact, the distinction that the environment provides honest interactions and the adversary provides malicious ones only exists in theory, but not in any concrete formalisation of the ITMs semantics. So, since both the adversary and the environment are ITMs that are $\forall$-quantified, it is sufficient to keep one entity to provide inputs and observe outputs (in this case the environment) and replace the other (i.e., the adversary) with a proxy.

The de-facto standard for UC proofs is to prove UC-emulation w.r.t. the dummy adversary and then apply Theorem 7 to obtain UC-emulation in the sense of Definition 2. Importing this theorem into the *RC* framework provides one key advantage for the proof of *RHP*. Once the simulator is defined, proving *RHP* amounts to proving trace equivalence of the compiled program and of the source program linked with the simulator. The key advantage is that these pieces of code are all defined, and so there is no need to perform any complicated induction and one can just reason about the semantics of the source and target programs. Even better, proving trace equivalence can be mechanised by using existing program analysis tools (as we do later with Deepsec in Section 7) that verify precisely that property.

As before, we first need to identify key axioms that regulate the behaviour (and the composition) of the dummy attacker in *RC* (Section 4.2.1) before proving Theorem 7 in *RC* (Section 4.2.2).

4.2.1 Additional Conditions for Dummy Attacker in RC. Borrowing from the proof to the dummy adversary theorem in UC, we rely on a *dummy program*.

Definition 10 (Dummy Program (informal))**.** The dummy program is a proxy that relays all messages from the environment to the program it is program-FFI composed with, and vice versa.

Intuitively, the dummy program is the dual of the dummy attacker from Definition 11. However, the dummy program only plays a role in the proof of the dummy attacker theorem and it does not appear anywhere else in any other formal statement. Likewise, in UC, the analogue notion appears only in the dummy adversary theorem, as part of the construction of distinguishing environment, but is never given a name.[8]

From a proof perspective, the dummy program appears as part of the attacker, as captured by Axiom 8. In fact, the Axiom essentially states that any attacker linked with a program ($\mathbf{A} \bowtie \mathbf{P}$) can be separated into an equivalent system with the same program, an attacker, and the dummy protocol ($\mathbf{A}' \bowtie (\mathbf{P_d} \circledast_{\mathbf{T}}^{\mathbf{T}} \mathbf{P})$). The second statement is essentially the same, but it allows the re-composition of an attacker ($\mathbf{A}$) and the dummy program ($\mathbf{P_d}$) into a single attacker (A').

Axiom 8 (Dummy Program Decomposition)**.** $\forall \mathbf{A}, \mathbf{P}, P.\ \exists \mathbf{A}', A'.$

$$\mathbf{A} \bowtie \mathbf{P} \eqsim \mathbf{A}' \bowtie (\mathbf{P_d} \circledast_{\mathbf{T}}^{\mathbf{T}} \mathbf{P}) \text{ and } \mathbf{A} \bowtie (\mathbf{P_d} \circledast_{S}^{\mathbf{T}} P) \eqsim A' \bowtie P$$

(Formalized in `dummy.prog_split` and `dummy.prog_unsplit`.)

Once the adversary part that is the dummy program has been identified, we can also carve out the part of the adversary that is the dummy attacker, as in Axiom 9.

Definition 11 (Dummy Attacker (informal))**.** In *RC*, the dummy attacker ($\mathbf{A_d}$) is a proxy that relays all messages from the environment to the attacker, and vice versa.

Axiom 9 (Dummy Attacker Decomposition)**.** $\forall \mathbf{A}, \mathbf{P}.\ \exists \mathbf{A}'.$

$$\mathbf{A} \bowtie (\mathbf{P_d} \circledast_{\mathbf{T}}^{\mathbf{T}} \mathbf{P}) \eqsim (\mathbf{A}' \circledcirc_{\mathbf{T}}^{\mathbf{T}} \mathbf{A_d}) \bowtie (\mathbf{P_d} \circledast_{\mathbf{T}}^{\mathbf{T}} \mathbf{P})$$

(Formalized in `dummy.adv_split`.)

Essentially, these two axioms state that it is possible to isolate the dummy program and the dummy attacker from an initial adversary, without altering the behaviour of the system.

4.2.2 Deriving the Dummy Attacker Theorem. Theorem 8 presents the dummy attacker theorem in *RC*.

Theorem 8 (Dummy Attacker in *RC*)**.**

$$\forall \mathbf{A}. \exists A.\ \mathbf{A} \bowtie [\![P]\!] \eqsim A \bowtie P \text{ iff } \exists A'.\ \mathbf{A_d} \bowtie [\![P]\!] \eqsim A' \bowtie P$$

Proof. (Proven in Isabelle/HOL: Theorem `dummy`) Figure 4 depicts the non-trivial direction of this proof ($\Leftarrow$) and how each of the presented axioms contribute to proving it (via transitivity of $\eqsim$). Below, we indicate assumption $\mathbf{A_d} \bowtie [\![P]\!] \eqsim A' \bowtie P$ with *HP*. One interesting bit is that this proof relies not just on the axioms presented in this section, but on two of the axioms presented in the previous one: Axiom 6 (FFI Recombination) and Axiom 7 (Constant Addition). □

We have now identified the set of axioms that allow a *RC* result to enjoy key UC properties such as the composition theorem and the dummy adversary one. Thus, the next section defines a language that fulfills those axioms (Section 5), in order to later on derive UC results as *RC* ones (Section 6).

[8] Not to be confused with UC's *dummy parties*, which have an entirely different purpose.

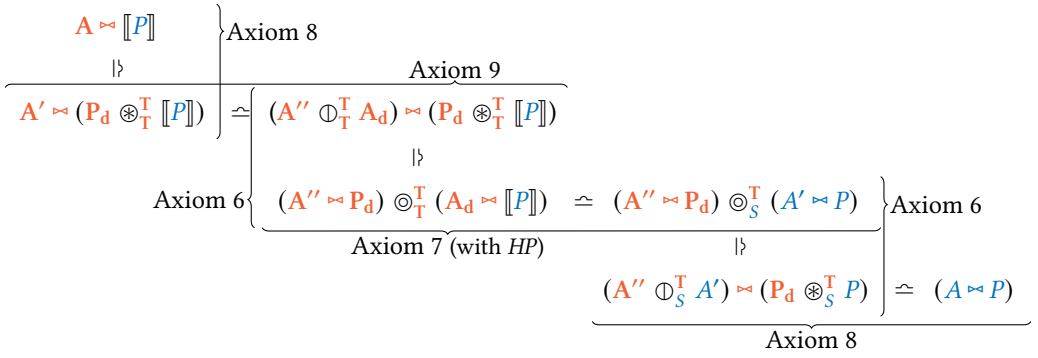

Fig. 4. Visual depiction of the proof of Theorem 8.

We believe both the composition theorem and the dummy attacker ones are crucial, and thus one should prove all of the presented axioms in whatever language one chooses for *RC*. However, note that these axioms are not necessary for the connection to hold, they merely provide additional useful results. In particular, the dummy attacker one simplifies the proofs to such a degree that mechanising them becomes possible using existing tools, as we show in Section 7.

5 THE REACTIVE, INTERACTIVE λ-CALCULUS (RILC)

This section presents the Reactive, Interactive λ-Calculus (RILC), a reactive language that we use to carry out UC proofs as *RC* ones. RILC extends the Interactive λ-calculus (ILC), so this section first presents the syntax, typing and operational semantics of ILC, alongside the key properties of the language (Section 5.1). Then this section presents the RILC-proper extensions: a trace model as required by *RC* and a module system to understand where each logical entity (e.g., protocol, attacker) ends (Section 5.2). Finally, this section showcases the properties of RILC, including the fact that it satisfies the axioms of Section 4 and thus it is possible to use it for our connection (Section 5.3).

As the name suggests, RILC is a *reactive* language. Reactive languages differ from non-reactive ones in one key element: they do not have a notion of whole program. In non-reactive languages, a program can run only if it is whole: all of its dependencies are resolved and there are no unknown function symbols. Instead, reactive languages have a built-in notion of a black-box environment that a program can both 'call' and 'be called' (or send a message to and receive a message from). The black box is therefore a catch-all element for all those symbols that are not resolved within the program.

Reactive languages are widely used in formal models for cryptographic protocol verification [18], and in our case they are the easiest kind of program for satisfying the axioms of Section 4. This is why we use a reactive language to instantiate our connection. However, we believe it is possible to instantiate the connection with non-reactive languages too, as we discuss in Section 8.3.

5.1 The Interactive λ-Calculus (ILC)

In a nutshell, ILC [73] is a process calculus that adapts ITMs to a subset of the π-calculus [91] through an affine typing discipline. To ensure that only one process is active (i.e., it can write) at any given time, processes implicitly pass around an affine "write token". When a process A writes to another process B, process A must have the write token, and by writing it passes the toke to process B, which now has the write token. Moreover, to maintain that the order of activations is

fully determined, the read endpoints of channels are (non-duplicable) affine resources, and so each write operation corresponds to a single, unique read operation. Together, these give ILC its central metatheoretic property of confluence.

This section recaps the elements of ILC we borrow from Liao et al. [73], namely its syntax (Section 5.1.1), its typing (Section 5.1.2) and its operational semantics (Section 5.1.3). Then, in order to familiarise the reader with ILC processes, it presents some shorthands for ILC notation that we use throughout this paper as well as an example ILC process (Section 5.1.4).

5.1.1 Syntax. The syntax of ILC is best described quoting from Liao et al. [73], starting from the types of the language.

$$
\begin{array}{ll}
\text{All Types} & U, V ::= A \mid X \\
\text{Sendable Types} & S, T ::= 1 \mid S \times S \mid S + S \\
\text{Unrestricted Types} & A, B ::= S \mid A \times A \mid A + A \mid \mathit{Wr}\ S \mid A \to_{\infty} U \mid A \to_{w} U \\
\text{Affine Types} & X, Y ::= !A \mid \mathit{Rd}\ S \mid X \otimes X \mid X \oplus X \mid X \to_{1} U
\end{array}
$$

Types (written U, V) are bifurcated into unrestricted types (written A, B) and affine types (written X, Y). A subset of the unrestricted types are sendable types (written S, T), i.e., the types of values that can be sent over channels. This restriction ensures that channels model network channels, which send only data. The sendable types include unit, products, and sums. The unrestricted types include the sendable types, products, sums, write endpoint types ($\mathit{Wr}\ S$), arrows, and write arrows ($A \to_w U$). Write arrows specify unrestricted abstractions for which the write token can be moved into the affine context of the abstraction body during reduction. The affine types include bang types, read endpoint types ($\mathit{Rd}\ S$), products, sums, and arrows.

$$
\begin{array}{llll}
\text{Labels} & \ell ::= \pi \mid w & \text{Multiplicity Labels} & \pi ::= 1 \mid \infty \\
\text{Channel Endpoint} & c ::= \mathit{Read}\,(d) \mid \mathit{Write}\,(d) & \text{Channel Names} & d \in \mathcal{D} \\
\text{Values} & \multicolumn{3}{l}{v ::= \mathit{unit} \mid \lambda_{\ell} x : U.\ e \mid \langle v, v \rangle_{\ell} \mid \mathit{inl}_{\ell}\ v \mid \mathit{inr}_{\ell}\ v \mid c \mid !v} \\
\text{Expressions} & \multicolumn{3}{l}{e ::= x \mid v \mid \langle e, e \rangle_{\ell} \mid \mathit{inl}_{\ell}\ e \mid \mathit{inr}_{\ell}\ e \mid (e\ e)_{\ell} \mid \mathit{fix}_{\ell}\,(x.e) \mid \mathit{let}_{\pi}\ x = e\ \mathit{in}\ e} \\
 & \multicolumn{3}{l}{\quad \mid \mathit{split}_{\ell}\,(e, x_1.x_2.e) \mid \mathit{case}_{\ell}\ e\ \mathit{of}\ \mathit{inl}\ x_1 \mapsto e \mid \mathit{inr}\ x_2 \mapsto e} \\
 & \multicolumn{3}{l}{\quad \mid\ !e \mid \imath e \mid v\,(x_1, x_2)\,.e \mid \mathit{wr}\,(e, e) \mid \mathit{rd}\,(e, x.e) \mid \mathit{ch}\,(e, x.e, e, x.e) \mid e\,(\|\ e)}
\end{array}
$$

For concision, certain syntactic forms are parameterized by a multiplicity π to distinguish between the unrestricted (∞) and affine (1) counterparts. Other syntactic forms are parameterized by a syntax label ℓ, which includes the multiplicity labels and the write label w (related to write effects). On introduction and elimination forms for functions (abstraction, application, and fixed points), the label w denotes variants that move around the write token as explained above. On introduction and elimination forms for products and sums, the label w denotes the sendable variants.

Values in ILC include unit, lambda expressions, pairs, sums, channel endpoints (written c), and banged values. We distinguish between the names of channel endpoints ($\mathit{Read}\,(d)$ and $\mathit{Write}\,(d)$) and the channel d itself that binds them. ILC supports a fairly standard feature set of expressions: pairs construction, left and right tagging, application, fixpoints, let-bindings, pair destruction, tag destruction, replication and un-replication, channel creation, writing and reading to channels, choice over multiple channel reads, and forking of a process.

5.1.2 Typing.

$$
\text{Typing Environment} \qquad \Gamma ::= \varnothing \mid \Gamma, (x : A)
$$

Affine Environment	$\Delta ::= \varnothing \mid \Delta, (x : X) \mid \Delta, \varpi$
Channel Environment	$\Psi ::= \varnothing \mid \Psi, (d : S)$

Typing relies on three environments: a classical typing environment that binds variables to unrestricted types, an affine environment that binds variables to affine types, and a channel environment binding channel names to sendable types. Notice that the write token ϖ lives in the affine context, though it cannot be bound to any variable. Instead, it flows around implicitly by virtue of where read and write effects are performed, as captured by the typing rules.

The typing judgement

$$\Psi; \Delta; \Gamma \vdash e : U$$

reads: *"under channel environment Ψ, affine environment Δ, and typing environment Γ, expression e has type U".*

Most typing rules are standard for process calculi and affine types, we report them below and refer the reader to Liao et al. [73] for an in-depth discussion of them.

$$\frac{\Psi(d) = S}{\Psi; \Delta; \Gamma \vdash \mathit{Read}\,(d) : \mathit{Rd}\ S}\ \text{(ILC-Type-rdend)} \qquad \frac{\Psi(d) = S}{\Psi; \Delta; \Gamma \vdash \mathit{Write}\,(d) : \mathit{Wr}\ S}\ \text{(ILC-Type-wrend)} \qquad \frac{}{\Psi; \Delta; \Gamma \vdash \mathit{unit} : 1}\ \text{(ILC-Type-unit)}$$

$$\frac{\Gamma(x) = A}{\Psi; \Delta; \Gamma \vdash x : A}\ \text{(ILC-Type-var-u)} \qquad \frac{\Delta(x) = X}{\Psi; \Delta; \Gamma \vdash x : X}\ \text{(ILC-Type-var-a)} \qquad \frac{\Psi; \Delta_1; \Gamma \vdash e_1 : A_1 \qquad \Psi; \Delta_2; \Gamma \vdash e_2 : A_2}{\Psi; \Delta_1, \Delta_2; \Gamma \vdash \langle e_1, e_2 \rangle_\infty : A_1 \times A_2}\ \text{(ILC-Type-pair-u)}$$

$$\frac{\Psi; \Delta_1; \Gamma \vdash e_1 : S_1 \qquad \Psi; \Delta_2; \Gamma \vdash e_2 : S_2}{\Psi; \Delta_1, \Delta_2; \Gamma \vdash \langle e_1, e_2 \rangle_w : S_1 \times S_2}\ \text{(ILC-Type-pair-s)} \qquad \frac{\Psi; \Delta_1; \Gamma \vdash e_1 : X_1 \qquad \Psi; \Delta_2; \Gamma \vdash e_2 : X_2}{\Psi; \Delta_1, \Delta_2; \Gamma \vdash \langle e_1, e_2 \rangle_1 : X_1 \otimes X_2}\ \text{(ILC-Type-pair-a)}$$

$$\frac{\Psi; \Delta; \Gamma \vdash e : A_1}{\Psi; \Delta; \Gamma \vdash \mathit{inl}_\infty\ e : A_1 + A_2}\ \text{(ILC-Type-inl-u)} \qquad \frac{\Psi; \Delta; \Gamma \vdash e : S_1}{\Psi; \Delta; \Gamma \vdash \mathit{inl}_w\ e : S_1 + A_2}\ \text{(ILC-Type-inl-s)}$$

$$\frac{\Psi; \Delta; \Gamma \vdash e : X_1}{\Psi; \Delta; \Gamma \vdash \mathit{inl}_1\ e : X_1 \oplus X_2}\ \text{(ILC-Type-inl-a)} \qquad \frac{\Psi; \Delta; \Gamma \vdash e : A_2}{\Psi; \Delta; \Gamma \vdash \mathit{inr}_\infty\ e : A_1 + A_2}\ \text{(ILC-Type-inr-u)}$$

$$\frac{\Psi; \Delta; \Gamma \vdash e : S_2}{\Psi; \Delta; \Gamma \vdash \mathit{inr}_w\ e : S_1 + S_2}\ \text{(ILC-Type-inr-s)} \qquad \frac{\Psi; \Delta; \Gamma \vdash e : X_2}{\Psi; \Delta; \Gamma \vdash \mathit{inr}_1\ e : X_1 \oplus X_2}\ \text{(ILC-Type-inr-a)}$$

$$\frac{\Psi; \Delta_1; \Gamma \vdash e : A_1 \times A_2 \qquad \Psi; \Delta_2; \Gamma, x_1 : A_1, x_2 : A_2 \vdash e' : U}{\Psi; \Delta_1, \Delta_2; \Gamma \vdash \mathit{split}_\infty\,(e, x_1.x_2.e') : U}\ \text{(ILC-Type-split-u)}$$

$$\frac{\Psi; \Delta_1; \Gamma \vdash e : S_1 \times S_2 \qquad \Psi; \Delta_2; \Gamma, x_1 : S_1, x_2 : S_2 \vdash e' : U}{\Psi; \Delta_1, \Delta_2; \Gamma \vdash \mathit{split}_w\,(e, x_1.x_2.e') : U}\ \text{(ILC-Type-split-s)}$$

$$\frac{\Psi; \Delta_1; \Gamma \vdash e : X_1 \times X_2 \qquad \Psi; \Delta_2, x_1 : X_1, x_2 : X_2; \Gamma \vdash e' : U}{\Psi; \Delta_1, \Delta_2; \Gamma \vdash \mathit{split}_1\,(e, x_1.x_2.e') : U}\ \text{(ILC-Type-split-a)}$$

$$\frac{\Psi; \Delta_1; \Gamma \vdash e : A_1 + A_2 \qquad \Psi; \Delta_2; \Gamma, x_1 : A_1 \vdash e_1 : U \qquad \Psi; \Delta_2; \Gamma, x_2 : A_2 \vdash e_2 : U}{\Psi; \Delta_1, \Delta_2; \Gamma \vdash \mathit{case}_\ell\ e\ \mathit{of}\ \mathit{inl}\ x_1 \mapsto e_1 \mid \mathit{inr}\ x_2 \mapsto e_2 : U}\ \text{(ILC-Type-case-u)}$$

$$\frac{\Psi; \Delta_1; \Gamma \vdash e : S_1 + S_2 \qquad \Psi; \Delta_2; \Gamma, x_1 : S_1 \vdash e_1 : U \qquad \Psi; \Delta_2; \Gamma, x_2 : S_2 \vdash e_2 : U}{\Psi; \Delta_1, \Delta_2; \Gamma \vdash \mathit{case}_w\ e\ \mathit{of}\ \mathit{inl}\ x_1 \mapsto e_1 \mid \mathit{inr}\ x_2 \mapsto e_2 : U}\ \text{(ILC-Type-case-s)}$$

$$\text{(ILC-Type-case-a)}\quad \frac{\Psi;\Delta_1;\Gamma \vdash e : X_1 \oplus X_2 \qquad \Psi;\Delta_2, x_1 : X_1;\Gamma \vdash e_1 : U \qquad \Psi;\Delta_2, x_2 : X_2;\Gamma \vdash e_2 : U}{\Psi;\Delta_1,\Delta_2;\Gamma \vdash case_1\ e\ of\ inl\ x_1 \mapsto e_1 \mid inr\ x_2 \mapsto e_2 : U}$$

$$\text{(ILC-Type-lam-u)}\quad \frac{\Psi;\varnothing;\Gamma, x : A \vdash e : U}{\Psi;\Delta;\Gamma \vdash \lambda_\infty x : A.\ e : A \to_\infty U} \qquad \text{(ILC-Type-lam-s)}\quad \frac{\Psi;\varnothing, Wr\ ;\Gamma, x : A \vdash e : U}{\Psi;\Delta;\Gamma \vdash \lambda_w x : A.\ e : A \to_w U}$$

$$\text{(ILC-Type-lam-a)}\quad \frac{\Psi;\Delta, x : X;\Gamma \vdash e : U}{\Psi;\Delta;\Gamma \vdash \lambda_1 x : X.\ e : X \to_1 U} \qquad \text{(ILC-Type-app-u)}\quad \frac{\Psi;\Delta_1;\Gamma \vdash e : A \to_\infty U \qquad \Psi;\Delta_2;\Gamma \vdash e' : A}{\Psi;\Delta_1,\Delta_2;\Gamma \vdash (e\ e')_\infty : U}$$

$$\text{(ILC-Type-app-s)}\quad \frac{\Psi;\Delta_1;\Gamma \vdash e : A \to_w U \qquad \Psi;\Delta_2;\Gamma \vdash e' : A}{\Psi;\Delta_1,\Delta_2, Wr\ ;\Gamma \vdash (e\ e')_w : U}$$

$$\text{(ILC-Type-app-a)}\quad \frac{\Psi;\Delta_1;\Gamma \vdash e : X \to_1 U \qquad \Psi;\Delta_2;\Gamma \vdash e' : X}{\Psi;\Delta_1,\Delta_2;\Gamma \vdash (e\ e')_1 : U} \qquad \text{(ILC-Type-fix-u)}\quad \frac{\Psi;\varnothing;\Gamma, x : A \to_\infty U \vdash e : A \to_\infty U}{\Psi;\Delta;\Gamma \vdash fix_\infty\ (x.e) : A \to_\infty U}$$

$$\text{(ILC-Type-fix-s)}\quad \frac{\Psi;\varnothing;\Gamma, x : A \to_w U \vdash e : A \to_w U}{\Psi;\Delta;\Gamma \vdash fix_w\ (x.e) : A \to_w U} \qquad \text{(ILC-Type-fix-a)}\quad \frac{\Psi;\varnothing, x : X \to_1 U;\Gamma \vdash e : X \to_1 U}{\Psi;\Delta;\Gamma \vdash fix_1\ (x.e) : X \to_1 U}$$

$$\text{(ILC-Type-let-u)}\quad \frac{\Psi;\Delta_1;\Gamma \vdash e : A \qquad \Psi;\Delta_2;\Gamma, x : A \vdash e : U}{\Psi;\Delta_1,\Delta_2;\Gamma \vdash let_\infty\ x = e\ in\ e' : U} \qquad \text{(ILC-Type-let-a)}\quad \frac{\Psi;\Delta_1;\Gamma \vdash e : X \qquad \Psi;\Delta_2, x : A;\Gamma \vdash e : U}{\Psi;\Delta_1,\Delta_2;\Gamma \vdash let_1\ x = e\ in\ e' : U}$$

$$\text{(ILC-Type-bang)}\quad \frac{\Psi;\Delta;\Gamma \vdash e : A}{\Psi;\Delta;\Gamma \vdash !e : !A} \qquad \text{(ILC-Type-gnab)}\quad \frac{\Psi;\Delta;\Gamma \vdash e : !A}{\Psi;\Delta;\Gamma \vdash ¡e : A} \qquad \text{(ILC-Type-nu)}\quad \frac{\Psi;\Delta, x_1 : Rd\ S;\Gamma, x_2 : Wr\ S \vdash e : U}{\Psi;\Delta;\Gamma \vdash \nu\ (x_1, x_2)\ .e : U}$$

$$\text{(ILC-Type-wr)}\quad \frac{\Psi;\Delta_1;\Gamma \vdash e : S \qquad \Psi;\Delta_2;\Gamma \vdash e' : Wr\ S}{\Psi;\Delta_1,\Delta_2,\varpi;\Gamma \vdash wr\ (e, e') : 1}$$

$$\text{(ILC-Type-rd)}\quad \frac{\Psi;\Delta_1;\Gamma \vdash e : Rd\ S \qquad \Psi;\Delta_2,\varpi, x :!S \otimes Rd\ S;\Gamma \vdash e' : U \qquad \varpi \notin \Delta_2}{\Psi;\Delta_1,\Delta_2;\Gamma \vdash rd\ (e, x.e') : U}$$

$$\text{(ILC-Type-choice)}\quad \frac{\begin{array}{c}\Psi;\Delta_1;\Gamma \vdash e_1 : Rd\ S \qquad \Psi;\Delta_2;\Gamma \vdash e_2 : Rd\ T \qquad \Psi;\Delta_3,\varpi, x :!S \otimes Rd\ S \otimes Rd\ T;\Gamma \vdash e_1' : U \\ \Psi;\Delta_3,\varpi, x :!T \otimes Rd\ S \otimes Rd\ T;\Gamma \vdash e_2' : U \qquad \varpi \notin \Delta_3\end{array}}{\Psi;\Delta_1,\Delta_2,\Delta_3;\Gamma \vdash ch\ \left(e_1, x.e_1', e_2, x.e_2'\right) : U}$$

$$\text{(ILC-Type-fork)}\quad \frac{\Psi;\Delta_1;\Gamma \vdash e : U \qquad \Psi;\Delta_2;\Gamma \vdash e' : V}{\Psi;\Delta_1,\Delta_2;\Gamma \vdash e\ (\|\ e') : U}$$

5.1.3 *Operational Semantics.*

Process Names	$p, q \in \mathcal{P}$	Names Sets	$\Sigma ::= \varnothing \mid \Sigma, d \mid \Sigma, p$
Process Pool	$\Pi ::= \varnothing \mid \Pi, p : e$	Configurations	$C ::= \langle \Sigma; \Pi \rangle$

$$\begin{aligned}\text{Evaluation Ctx.}\quad E ::= {} & [\cdot] \mid \langle E, e\rangle_\ell \mid \langle v, E\rangle_\ell \mid inl_\ell\ E \mid inr_\ell\ E \mid (E\ e)_\ell \mid (v\ E)_\ell \mid let_\pi\ x = E\ in\ e \\ & \mid split_\ell\ (E, x_1.x_2.e) \mid case_\ell\ E\ of\ inl\ x_1 \mapsto e \mid inr\ x_2 \mapsto e \mid !E \mid ¡E \\ & \mid wr\ (E, e) \mid wr\ (v, E) \mid rd\ (E, x.e) \mid ch\ (E, x.e, e, x.e) \mid ch\ (e, x.e, E, x.e)\end{aligned}$$

The operational semantics of ILC relies on the notion of configuration C, which is a tuple of (existing) channel and process names Σ, and a pool of running and terminated processes Π. The operational

semantics of ILC is a small-step concurrent semantics that relies on a contextual semantics for the reduction of expressions. The following judgements comprise the operational semantics of ILC:

$C \equiv C'$	congruence rules between configurations
$c \rightsquigarrow c'$	channel endpoint communication: c communicates to c'
$C \hookrightarrow C'$	configuration reduction: C takes a step and becomes C'
$e \hookrightarrow_0 e'$	expression primitive reduction: e takes a primitive step and becomes e'

The rules of the operational semantics are standard for process calculi; we report them below and refer the reader to Liao et al. [73] for an in-depth discussion of them.

(ILC-eq-conf)
$$\frac{\Pi \equiv_{perm} \Pi'}{\langle \Sigma; \Pi \rangle \equiv \langle \Sigma; \Pi' \rangle}$$

(ILC-connect)
$$\frac{}{Wr\ d \rightsquigarrow Rd\ d}$$

(ILC-Sem-local)
$$\frac{e \hookrightarrow_0 e'}{\langle \Sigma; \Pi, p : E\,[e] \rangle \hookrightarrow \langle \Sigma; \Pi, p : E\,[e'] \rangle}$$

(ILC-Sem-fork)
$$\frac{q \notin \Sigma}{\langle \Sigma; \Pi, p : E\,[e\,(\parallel e')] \rangle \hookrightarrow \langle \Sigma, q; \Pi, p : E\,[e]\,, q : e' \rangle}$$

(ILC-Sem-nu)
$$\frac{d \notin \Sigma}{\langle \Sigma; \Pi, p : E\,[v\,(x_1, x_2)\,.e] \rangle \hookrightarrow \langle \Sigma, d; \Pi, p : E\,[e[Read\,(d)\ /\ x_1][Write\,(d)\ /\ x_2]] \rangle}$$

(ILC-Sem-read-write)
$$\frac{c_2 \rightsquigarrow c_1}{\begin{array}{l}\langle \Sigma; \Pi, p : E\,[rd\,(c_1, x.e)]\,, q : E'\,[wr\,(v, c_2)] \rangle \hookrightarrow \\ \qquad \langle \Sigma; \Pi, p : E\,[e[\langle !v, c_1 \rangle_1\ /\ x]]\,, q : E'\,[unit] \rangle\end{array}}$$

(ILC-Sem-choice)
$$\frac{c \rightsquigarrow c_i \text{ for } i \in 1..2}{\begin{array}{l}\langle \Sigma; \Pi, p : E\,[ch\,(c_1, x_1.e_1, c_2, x_2.e_2)]\,, q : E'\,[wr\,(v, c)] \rangle \hookrightarrow \\ \qquad \langle \Sigma; \Pi, p : E\,[e_i[\langle !v, c_1, c_2 \rangle_1\ /\ x_i]]\,, q : E'\,[unit] \rangle\end{array}}$$

(ILC-Sem-cong)
$$\frac{C_1 \equiv C_1' \qquad C_1' \hookrightarrow C_2' \qquad C_2' \equiv C_2}{C_1 \hookrightarrow C_2}$$

(ILC-Sem-ex-letin)
$$\frac{}{let_\pi\ x = v\ in\ e \hookrightarrow_0 e[v\ /\ x]}$$

(ILC-Sem-ex-beta)
$$\frac{}{(\lambda_\ell x.\ e\ v)_\ell \hookrightarrow_0 e[v\ /\ x]}$$

(ILC-Sem-ex-split)
$$\frac{}{split_\ell\,(\langle v_1, v_2 \rangle_\ell\,, x_1.x_2.e) \hookrightarrow_0 e[v_1\ /\ x_1][v_2\ /\ x_2]}$$

(ILC-Sem-ex-case-l)
$$\frac{}{case_\ell\ inl_\ell\ v\ of\ inl\ x_1 \mapsto e \mid inr\ x_2 \mapsto e' \hookrightarrow_0 e[v\ /\ x_1]}$$

(ILC-Sem-ex-case-r)

$$\frac{}{case_\ell\ inr_\ell\ v\ of\ inl\ x_1 \mapsto e \mid inr\ x_2 \mapsto e' \ \hookrightarrow_0\ e'[v\ /\ x_2]}$$

(ILC-Sem-ex-fix)

$$\frac{}{fix_\ell\ (x.e)\ \hookrightarrow_0\ e\left[fix_\ell\ (x.e)\ /\ x\right]}$$

(ILC-Sem-ex-gnabang)

$$\frac{}{¡!v\ \hookrightarrow_0\ v}$$

5.1.4 Shorthands and ILC Example. The syntax of ILC lacks many useful real-world shorthands. In this section we present the semantics for the following such useful shorthands:

- $fwd(Read\,(d), Write\,(d'))$ defines a proxy that forwards all messages read on channel d to channel d' (Rule ILC-eq-fwd);
- $\otimes$ models the usual binary operations (+,−,==, etc.) on values (Rule ILC-Sem-ex-bop);
- Rules ILC-Sem-ex-ifte-t and ILC-Sem-ex-ifte-f show if-then-else reductions that can be obtained from case destruction of tagged values;
- $loop(e)$ models an unbound loop (Rule ILC-Sem-ex-loop);
- $loop(n)(e)$ models an bounded loop that runs for n iterations (Rules ILC-Sem-ex-loopN and ILC-Sem-ex-loop1).

(ILC-eq-fwd)

$$\frac{}{\langle \Sigma; \Pi, p : E\,[fwd(Read\,(d), Write\,(d'))]\rangle \equiv \langle \Sigma; \Pi, p : E\,[rd\,(Read\,(d), x.wr\,(x, Write\,(d')))]\rangle}$$

(ILC-Sem-ex-bop)

$$\frac{v'' = \otimes(v, v')}{v \otimes v' \ \hookrightarrow_0\ v''}$$

(ILC-Sem-ex-ifte-t)

$$\frac{}{if\ true\ then\ e\ else\ e' \ \hookrightarrow_0\ e}$$

(ILC-Sem-ex-ifte-f)

$$\frac{}{if\ false\ then\ e\ else\ e' \ \hookrightarrow_0\ e'}$$

(ILC-Sem-ex-loop)

$$\frac{}{loop(e)\ \hookrightarrow_0\ e; loop(e)}$$

(ILC-Sem-ex-loopN)

$$\frac{n > 1}{loop(n)(e)\ \hookrightarrow_0\ e; loop(n-1)(e)}$$

(ILC-Sem-ex-loop1)

$$\frac{}{loop(1)(e)\ \hookrightarrow_0\ e}$$

We defer presenting an example of ILC code (and its semantics) to Section 6. Note that, in the examples, we often massage the ILC syntax to enhance readability. For example, we will write code in an imperative form (more similar to other similar languages used by tools that perform protocol verification). Moreover, we will write *wr* (*Msg v*) *from Write*(*PQ*) instead of writing *wr* (*Msg v*, *Write* (*PQ*)), assuming that messages carry a defining header (in this case, *Msg*). Finally, we will write *rd* (*Msg x*) *from Read*(*PQ*) instead of writing a more complex read expression that reads on channel end *Read* (*PQ*) and then checks that the read value matches some header *Msg* and binds the read value in *x*.

5.2 RILC: Modules, Traces and Cryptographic Additions to ILC

This section presents our addition to ILC that define RILC. The first such addition is a series of cryptographic primitives that are used to describe the hash-based commitment protocol of Section 6 (Section 5.2.1).

The second one is a module system that is used to clearly identify the different modules (read, protocols and sub-protocols) that comprise a RILC program (Section 5.2.2).

The final addition is a reactive trace semantics whose purpose is twofold (Section 5.2.3). First, the reactive semantics lifts the language into a reactive setting, where RILC programs (now called modules) interact with an unspecified environment. Second, the traces capture any interaction that happens on the interface between modules and the environment.

5.2.1 Cryptographic Additions. Vanilla ILC does not contain cryptographic primitives, but the authors provide some additions to ILC [73, Appendix] in order to model a commitment protocol (the same protocol we also model later). In this version of RILC, we add cryptographic primitives

for symbolic pseudorandom generation with a trapdoor and xor-ing of symbolic values. Our cryptographic additions are handled in a separate part of the program state, so that they can be easily extended and replaced with minimal effort for ensuring their additions do not violate any metatheoretical property.

$$
\begin{array}{llll}
\text{Security Parameter} & \lambda \in \mathbb{N} & \text{Random Seed} & n^{rnd} ::= b_1 \cdots b_j \\
\text{Seals} & \sigma \in \mathcal{S} & \text{Values} & v ::= \cdots \mid \sigma \\
\text{Expressions} & e ::= \cdots \mid \mathit{takernd} \mid \mathit{keygen}\,(e) \mid \mathit{prg}\,(e, e) \mid \mathit{invert}\,(e, e) \mid \mathit{xors}(e, e) & & \\
\text{Evaluation Contexts} & E ::= \cdots \mid \mathit{keygen}\,(E) \mid \mathit{prg}\,(E, e) \mid \mathit{prg}\,(v, E) \mid \mathit{invert}\,(E, e) & & \\
 & \quad \mid \mathit{invert}\,(v, E) \mid \mathit{xors}(E, e) \mid \mathit{xors}(v, E) & &
\end{array}
$$

To model the output of a symbolic prg, we introduce the notion of seals σ, which are values and opaque tokens (which have been used to model other cryptographic primitives such as additions in other work [9, 93, 94]. There are a number of expressions that deal with cryptographic primitives: taking a λ-long random bitstring, generating a key, prg-hashing a value with a key, inverting the result of a hash via a trapdoor, and xor-ing values. In general, we assume a security parameter λ which is an integer and a randomness parameter n^{rnd}, which is a string of bits whose length is a polynomial function of λ.

Typing of Cryptographic Additions. Typing cryptographic additions is straightforward (and therefore omitted), since they do not pass the write token around. The only meaningful change to the typing is that *takernd* needs to know the length of the security parameter in order to tell the type of the bitstring it returns. Thus, typing rules need to be extended to carry around λ. Since this is a minor, tedious addition that mostly clutters the notation, we elide that annotation in subsequent judgements.

Operational Semantics of Cryptographic Additions.

$$
\begin{array}{ll}
\text{Cryptographic Stores} & G ::= \varnothing \mid G, \mathit{xor}\,(\sigma, v, v) \mid G, \langle v, v', \langle \sigma, \bot, \bot \rangle \rangle \mid G, \langle v, v', \langle \sigma, v'', \sigma' \rangle \rangle \\
\text{Leaked Knowledge} & H ::= \varnothing \mid H, v \\
\text{Cryptographic Configuration} & K ::= \left\langle H; G; \lambda; n^{rnd} \right\rangle
\end{array}
$$

In order to present the semantics of the cryptographic expressions, we need to define additional elements of the runtime state that the operational semantics rules rely on.

Cryptographic stores G are a store needed to collect any information required by the cryptographic additions. In this work, the store contains: the result of *xors* operations, bindings for a random number v and a fresh key v' bound to a trapdoor σ that has not been used (thus the two $\bot$), and bindings for a key v used to create the prg σ' for a value v''. Leaked knowledge represents all values that have been leaked, the specific way in which leakage takes place (i.e., how modules can leak data to the attacker) is presented later, in Section 5.2.3. Cryptographic configurations K are the part of a runtime state that contains all cryptography-related additions.

The operational semantics that handles cryptographic additions follows these judgements

$$
\begin{array}{ll}
K \triangleright e \hookrightarrow_0 K' \triangleright e' & \text{Primitive reduction relying on the cryptographic configuration} \\
\langle K, C \rangle \hookrightarrow \langle K', C' \rangle & \text{Cryptographic configuration takes a step}
\end{array}
$$

Below are the semantics rules of the cryptographic additions.

$$\frac{\begin{array}{c}K = \langle H; G; \lambda; n^{rnd}\rangle \quad n^{rnd} = b_0 \cdots b_{\lambda-1} \cdots b_j \quad \lambda - 1 \leq j \\ n' = b_0 \cdots b_{\lambda-1} \quad n_c^{rnd} = b_\lambda \cdots b_j \quad K' = \langle H; G; \lambda; n_c^{rnd}\rangle\end{array}}{K \triangleright takernd \hookrightarrow_0 K' \triangleright n'} \text{(RILC-Sem-take)}$$

$$\frac{\begin{array}{c}K = \langle H; G; \lambda; n^{rnd}\rangle \quad G' = G, \langle v_{rand}, v_{pubkey}, \langle \sigma_{trap}, \bot, \bot\rangle\rangle \quad \texttt{fresh}\,(v_{rand}, G) \\ \texttt{fresh}\,(v_{pubkey}, G) \quad \texttt{fresh}\,(\sigma_{trap}, G) \quad K' = \langle H; G'; \lambda; n^{rnd}\rangle\end{array}}{K \triangleright keygen\,(v_{rand}) \hookrightarrow_0 K' \triangleright \langle v_{pubkey}, \sigma_{trap}\rangle_\infty} \text{(RILC-Sem-keygen)}$$

$$\frac{K = \langle H; G; \lambda; n^{rnd}\rangle \quad \langle v_{rand}, v_{pubkey}, \langle \sigma_{trap}, \bot, \bot\rangle\rangle \in G}{K \triangleright keygen\,(v_{rand}) \hookrightarrow_0 K \triangleright \langle v_{pubkey}, \sigma_{trap}\rangle_\infty} \text{(RILC-Sem-keygen)}$$

$$\frac{\begin{array}{c}K = \langle H; G; \lambda; n^{rnd}\rangle \quad \texttt{fresh}\,(\sigma, G) \quad \langle v_{rand}, v_{pubkey}, \langle \sigma_{trap}, \bot, \bot\rangle\rangle \in G \\ G' = G, \langle v_{rand}, v_{pubkey}, \langle \sigma_{trap}, v_{in}, \sigma\rangle\rangle \setminus \langle v_{rand}, v_{pubkey}, \langle \sigma_{trap}, \bot, \bot\rangle\rangle \quad K' = \langle H; G'; \lambda; n^{rnd}\rangle\end{array}}{K \triangleright prg\,(v_{pubkey}, v_{in}) \hookrightarrow_0 K' \triangleright \sigma} \text{(RILC-Sem-trapdoor)}$$

$$\frac{K = \langle H; G; \lambda; n^{rnd}\rangle \quad \langle v_{rand}, v_{pubkey}, \langle \sigma_{trap}, v_{in}, \sigma\rangle\rangle \in G}{K \triangleright prg\,(v_{pubkey}, v_{in}) \hookrightarrow_0 K \triangleright \sigma} \text{(RILC-Sem-trapdoor-exists)}$$

$$\frac{K = \langle H; G; \lambda; n^{rnd}\rangle \quad \exists v_{in}.\, \langle v_{rand}, v_{pubkey}, \langle \sigma_{trap}, v_{in}, \sigma\rangle\rangle \in G}{K \triangleright invert\left(\langle v_{pubkey}, \sigma_{trap}\rangle_\infty, \sigma\right) \hookrightarrow_0 K \triangleright true} \text{(RILC-Sem-invert-true)}$$

$$\frac{K = \langle H; G; \lambda; n^{rnd}\rangle \quad \nexists v_{in}.\, \langle v_{rand}, v_{pubkey}, \langle \sigma_{trap}, v_{in}, \sigma\rangle\rangle \in G}{K \triangleright invert\left(\langle v_{pubkey}, \sigma_{trap}\rangle_\infty, \sigma\right) \hookrightarrow_0 K \triangleright false} \text{(RILC-Sem-invert-false)}$$

$$\frac{\begin{array}{c}K = \langle H; G; \lambda; n^{rnd}\rangle \quad K' = \langle H; G'; \lambda; n^{rnd}\rangle \\ \text{if } xor\,(v, \_, \_) \notin G \text{ then } G' = G, xor\,(v, v', \sigma) \text{ and } \texttt{fresh}\,(\sigma, G) \\ \text{if } xor\,(v, v', \sigma) \in G \text{ then } G' = G \\ \text{if } xor\,(v, v'', \sigma'') \in G \text{ and } v' \neq v'' \text{ then } G' = G, xor\,(v, v', \sigma) \text{ and } \texttt{fresh}\,(\sigma, G)\end{array}}{K \triangleright xors(v, v') \hookrightarrow_0 K' \triangleright \sigma} \text{(RILC-Sem-xor-seal)}$$

$$\frac{K \triangleright e \hookrightarrow_0 K' \triangleright e'}{\langle K; \Sigma; \Pi, p : E\,[e]\rangle \hookrightarrow \langle K'; \Sigma; \Pi, p : E\,[e']\rangle} \text{(RILC-Sem-crypto)} \qquad \frac{\langle C\rangle \hookrightarrow \langle C\rangle}{\langle K; C\rangle \hookrightarrow \langle K; C\rangle} \text{(RILC-Sem-local)}$$

Rule RILC-Sem-take takes the first λ-long bitstring from the randomness bits n^{rnd} and updates that accordingly. Rule RILC-Sem-keygen takes a random value v_{rand} and generates a fresh binding for a key v_{pubkey} with a known trapdoor σ_{trap}. Rule RILC-Sem-keygen returns the existing binding in case the key generation is called with a known input. Rule RILC-Sem-trapdoor calculates the fresh prg σ of a value v_{in} with some key v_{pubkey}. Rule RILC-Sem-trapdoor-exists is used in case the prg of some value and some key has already been calculated, in which case, the result is fetched from the cryptographic store G. Rule RILC-Sem-invert-true acknowledges a correct inversion of a prg σ given its key v_{pubkey} and the trapdoor σ_{trap}, while Rule RILC-Sem-invert-false is used when

the inversion is used on a prg with either the wrong key or trapdoor. Rule RILC-Sem-xor-seal models a simple, one-directional exclusive or.

Rule RILC-Sem-crypto lifts the cryptographic primitive reductions to the cryptographic state while Rule RILC-Sem-local lifts any previous reduction from Section 5.1.3 that does not use cryptographic primitives to the cryptographic state.

In the following, we define the initial cryptographic state via function $K_0(\cdot)$, which takes a security parameter and a random bitstring in input and returns the empty cryptographic state.

$$K_0(\lambda, n^{rnd}) \stackrel{\text{def}}{=} \langle \varnothing, \varnothing, \lambda, n^{rnd} \rangle$$

Cryptography for the Environment. As we have hinted (but will only show in Section 5.2.3), RILC is a reactive language, whose model contains a black-box environment that can send messages to the reactive program being run. The cryptographic additions should also let the semantics calculate what is the environment allowed to compute, and this is what the cryptographic environment semantics describes. The judgement for this semantics is:

$\vdash_{env} K \rightsquigarrow K' \triangleright v$ — From knowledge K, the environment produces value v and augments its knowledge to K'

Below are the rules for this judgement, which we keep to a minimum for simplicity.

$$\frac{v \neq \sigma}{\vdash_{env} K \rightsquigarrow K \triangleright v}\ \text{(RILC-Sem-EC-base)} \qquad \frac{K = \langle H; G; \lambda; n^{rnd} \rangle \quad v \in H}{\vdash_{env} K \rightsquigarrow K \triangleright v}\ \text{(RILC-Sem-EC-history)} \qquad \frac{K \triangleright takernd \hookrightarrow_0 K' \triangleright n'}{\vdash_{env} K \rightsquigarrow K' \triangleright n'}\ \text{(RILC-Sem-EC-take)}$$

$$\frac{\vdash_{env} K \rightsquigarrow K' \triangleright v \quad \vdash_{env} K' \rightsquigarrow K'' \triangleright v' \quad K'' \triangleright xors(v, v') \hookrightarrow_0 K''' \triangleright \sigma \quad K''' = \langle H; G; \lambda; n^{rnd} \rangle \quad K_f = \langle H, \sigma; G; \lambda; n^{rnd} \rangle}{\vdash_{env} K \rightsquigarrow K_f \triangleright \sigma}\ \text{(RILC-Sem-EC-xor)}$$

$$\frac{\vdash_{env} K \rightsquigarrow K' \triangleright v \quad K' \triangleright keygen(v) \hookrightarrow_0 K'' \triangleright \langle v_{pubkey}, \sigma_{trap} \rangle_\infty \quad K'' = \langle H; G; \lambda; n^{rnd} \rangle \quad K_f = \langle H, \sigma_{trap}, v_{pubkey}; G; \lambda; n^{rnd} \rangle}{\vdash_{env} K \rightsquigarrow K_f \triangleright \langle v_{pubkey}, \sigma_{trap} \rangle_\infty}\ \text{(RILC-Sem-EC-keygen)}$$

$$\frac{\vdash_{env} K \rightsquigarrow K' \triangleright v \quad \vdash_{env} K' \rightsquigarrow K'' \triangleright v' \quad K'' \triangleright prg(v, v') \hookrightarrow_0 K''' \triangleright \sigma \quad K''' = \langle H; G; \lambda; n^{rnd} \rangle \quad K_f = \langle H, \sigma; G; \lambda; n^{rnd} \rangle}{\vdash_{env} K \rightsquigarrow K_f \triangleright \sigma}\ \text{(RILC-Sem-EC-prg)}$$

An environment can generate any value that is a ground value (Rule RILC-Sem-EC-base), any value that has been sent to it in the past (Rule RILC-Sem-EC-history), or any random value from the randomness bitstring (Rule RILC-Sem-EC-take). An environment can also compute the xor of any value it could generate (Rule RILC-Sem-EC-xor) and generate a fresh key (Rule RILC-Sem-EC-keygen), an act that leaks the key trapdoor to the environment knowledge. Finally, an environment can create a fresh prg, so long as the values used in its computation were known to it (Rule RILC-Sem-EC-prg). Given that our treatment of cryptographic elements is symbolic, we do not concern ourselves with modelling a semantics with a computational bound for the environment. We believe lifting this simplification can be done by relying on existing work, e.g., [77].

5.2.2 *A Module System.*

Modules $M ::= (D; \Pi; I; X)$

Endpoint Types	$\tau ::= Rd\ S \mid Wr\ S$
Imports, Exports, Definitions	$I, X, D ::= \varnothing \mid I, c : \tau$
Processes	$\Pi ::= \varnothing \mid \Pi, p : rd\,(Read\,(d), x.e')$

The key element of a module system is the definition of modules (M), which are a tuple consisting of defined channel names, process definitions, imports and exports. First, any top-level process starts by reading from a channel, and thus we change the top-level definition of processes accordingly. For well-typedness reasons, that top-level read channel, alongside any other channel used in the process of the body form the definitions D of a module. Imports define the dependencies of the module: those channel names are not defined by the module and thus external modules (or the environment) will provide an implementation for them. Exports define what a module supplies for external modules to use, essentially any channel name that is defined but not exported is local to the module. Any channel end c that appears in definitions, imports or exports carries its endpoint type, meaning, it describes what kind of channel end is and how it is supposed to be used.

In the following, for simplicity, with a slight abuse of notation, we assume that these types mention sessions, i.e., the full sequence of the type of messages that are to be exchanged on the channel. Note that it would be possible to simply use a different channel after each message is communicated, but this would make many an example hard to follow. We leave a full formalisation of session types for RILC for future work.

The operation that happens on modules is linking, denoted with $M_1 \bowtie M_2$ (Rule RILC-Linking).

(RILC-Linking)

$$\frac{M_1 = D_1; \Pi_1; I_1; X_1 \qquad M_2 = D_2; \Pi_2; I_2; X_2 \qquad I = I_1 \setminus X_2 \cup I_2 \setminus X_1 \qquad X = X_1 \setminus I_2 \cup X_2 \setminus I_1 \qquad D = D_1 \cup D_2 \qquad \Pi = \Pi_1 \cup \Pi_2}{M_1 \bowtie M_2 = D; \Pi; I; X}$$

Module linking does what one expects, it generates another module whose processes and definitions are the union of the processes and definitions of its sub-modules. Then, the resulting module has all the imports and exports of its sub-modules, minus those that are fulfilled by the other module. That is, the resulting module has all imports of the two sub-modules, minus their exports, and all the exports of the two sub-modules, minus their imports. When performing the set difference (as in $I_1 \setminus X_2$), we assume that subtracting a write (resp. read) end for a channel remove the respective read (resp. write) end from the set, i.e., removing $Write\,(d)$ from $\{Read\,(d), Read\,(e)\}$ results in $\{Read\,(e)\}$.

Note on Notation: From the type of linking ($\bowtie : M \times M \to M$), it is evident that the top-level notion of partial programs that is of interest is indeed modules. In the previous part of the paper, we used metavariable P to range over such partial programs. In the case of RILC specifically, we use M to indicate what was previously denoted with P. We do not rebind the metavariable P for RILC, which can be then thought as an alternative to M.

Typing of Modules. Modules are typed according to these new judgements.

$\vdash M$	Well-typed module
$D, I \vdash \Pi$	Well-typed process with its defined channels and imports

The typing of modules is straightforward (Rule RILC-Type-Module). A module is well-typed if its processes are, according to the process typing (Rule RILC-Type-Process), which in turns relies on the expression typing of Section 5.1.2. Note that module and process typing do not tamper with the write token, nor with the affinity of resources, and thus their addition easily preserve the properties of ILC.

$$\frac{M = (D; \Pi; I; X) \quad \Pi = \Pi_1, \ldots, \Pi_n \quad \bigcap_{i \in 1..n} \Pi_i.p = \varnothing \quad D = \bigcup_{i \in 1..n} D_i \quad I = \bigcup_{i \in 1..n} I_i \quad X \subseteq D \quad \forall i \in 1..n.\ D; I \vdash \Pi_i}{\vdash M} \text{ (RILC-Type-Module)}$$

$$\frac{\Pi = p : rd\,(Read\,(d), x.e') \quad Read\,(d) : Rd\ S \in D \quad D; \varnothing; \varnothing \vdash Read\,(d) : Rd\ S \quad D, I; \varnothing; \varnothing \vdash rd\,(Read\,(d), x.e) : U}{D; I \vdash \Pi} \text{ (RILC-Type-Process)}$$

With a slight abuse of notation we write D and I to mean their channel and types that are used to create the Ψ used for typing expressions in the premise of Rule RILC-Type-Process.

5.2.3 A Reactive Trace Semantics. The reactive trace semantics that makes RILC a reactive language relies on two key notions: reactive programs and traces. A reactive program contains some internal state, which is not directly observable from an external observer and reacts to a sequence of inputs from an environment by producing a sequence of observable outputs. After each input, the program may update its internal state, allowing all past inputs to influence an output. By definition, a reactive program is really a component of a larger system that provides it inputs. In RILC, the reactive program is a module (and we use the two words interchangeably in this section) and the larger system is an unspecified black-box that represents the classical UC environment. The input and output sequences of messages exchanged between the reactive program and the environment are what constitute an interaction trace (from Section 2).

Reactive Configurations	$\Omega ::= \langle \eta; K; \Xi; C \rangle$
Token Tag	$\eta ::= \varpi \mid \bot$
External Names	$\Xi ::= \varnothing \mid \Xi, c : S$

From an operational semantics perspective, dealing with reactive programs changes the notion of configuration, which now becomes a reactive configuration Ω. A reactive configuration keeps track of the write token in the token tag η, in order to know whether the reactive program ($\bot$) or the environment (ϖ) has the right to write. Then, it also keeps track of the cryptographic store K, as presented in Section 5.2.1. The reactive configuration then contains a list of external names Ξ, that is, a list of those names that the reactive program relies on and that the environment is supposed to handle the other end of. Finally, the reactive configuration contains the elements of an ILC configuration from Section 5.1.3, namely the existing names Σ and the process pool Π.

Actions	$\tau ::= \text{send } v \text{ on } c? \mid \text{send } v \text{ on } c! \mid \epsilon$	Interaction Traces	$\overline{\tau} ::= \varnothing \mid \tau \cdot \overline{\tau}$
Probabilities	$\rho \in 0..1$	Traces	$\overline{t} ::= (\rho, \overline{\tau})$

Traces $\overline{t}$ are pairs of a probability ρ and an interaction trace $\overline{\tau}$. An interaction trace is a sequence of actions τ. Actions include: the environment sending a message v on channel c to the reactive program (send v on c?), the reactive program sending a message v on channel c to the environment (send v on c!), and a silent action ϵ. The actions use ? and ! decorations borrowed from the process calculi literature [91] to indicate the "direction" of the message. Note that the semantics will insist that interaction traces start with a ? action and then alternate between ! and ? ones.

Operational Reactive Trace Semantics. The semantics relies on the following judgements, where we recap all previous judgements too, for the sake of completeness.

$\Omega \equiv \Omega'$	Configurations are equivalent, as in Section 5.1.3
$c \rightsquigarrow c'$	c writes where c' reads, as in Section 5.1.3
$e \hookrightarrow_0 e'$	Expression reduction, as in Section 5.1.3
$C \hookrightarrow C'$	Configuration takes a step, as in Section 5.1.3
$\langle K, C\rangle \hookrightarrow \langle K, C'\rangle$	Crypto configuration takes a step, as in Section 5.2.1
$\Omega \overset{\tau}{\hookrightarrow} \Omega'$	Reactive configuration takes a step with action τ
$\Omega \overset{\overline{\tau}}{\Longrightarrow} \Omega'$	Reactive configuration big-steps with interaction trace $\overline{\tau}$

Many of the judgements of the semantics we rely on are untouched from previous definitions. Below are those rules of the semantics that deal with the robustness and the traces.

$$\text{(RILC-Sem-Write)}\quad \frac{c \in \Xi \qquad K = \left\langle H; G; \lambda; n^{rnd}\right\rangle \qquad K' = \left\langle H, v; G; \lambda; n^{rnd}\right\rangle}{\langle \bot; K; \Xi; \Sigma; \Pi, p : E\,[wr\,(v, c)]\rangle \xrightarrow{\text{send } v \text{ on } c!} \langle \varpi; K'; \Xi; \Sigma; \Pi, p : E\,[unit]\rangle}$$

$$\text{(RILC-Sem-Read)}\quad \frac{c \in \Xi \qquad \vdash_{env} K \rightsquigarrow K' \triangleright v}{\langle \varpi; K; \Xi; \Sigma; \Pi, p : E\,[rd\,(c, x.e)]\rangle \xrightarrow{\text{send } v \text{ on } c?} \langle \bot; K'; \Xi; \Sigma; \Pi, p : E\,[e[\langle v, c\rangle \,/\, x]]\rangle}$$

$$\text{(RILC-Sem-choice)}\quad \frac{c_i \in \Xi \qquad \vdash_{env} K \rightsquigarrow K' \triangleright v}{\begin{array}{l}\langle \varpi; K; \Xi; \Sigma; \Pi, p : E\,[ch\,(c_1, x_1.e_1, c_2, x_2.e_2)]\rangle \hookrightarrow \\ \qquad \langle \bot; K'; \Xi; \Sigma; \Pi, p : E\,[e_i[\langle !v, c_1, c_2\rangle_1 \,/\, x_i]]\rangle\end{array}}$$

$$\text{(RILC-Sem-Other)}\quad \frac{\langle K; \Sigma; \Pi\rangle \hookrightarrow \langle K'; \Sigma; \Pi'\rangle}{\langle \bot; K; \Xi; \Sigma; \Pi\rangle \xrightarrow{\epsilon} \langle \bot; K'; \Xi; \Sigma; \Pi'\rangle}$$

$$\text{(RILC-Sem-Error)}\quad \frac{}{\langle \bot; K; \Xi; \Sigma; \Pi, p : E\,[error]\rangle \xrightarrow{\Lightarrow} \langle \varpi; K; \Xi; \Sigma; \varnothing\rangle}$$

Rule RILC-Sem-Write states that when the reactive program has the write token (η=$\bot$), and it is writing on a channel c that belongs to the environment ($c \in \Xi$), then the environment can read the value that is being written, and that value is added to the environment knowledge K'. This is the way leakage (as left hanging from Section 5.2.1) is generated: values that are written into channels that belong to external names are effectively leaked to the environment (that is, to the attacker).

Dually, Rule RILC-Sem-Read states that when the environment has the write token ($\eta = \varpi$), and the reactive program is reading on a channel c that belongs to the environment, then the environment can send a value v that the program reads. The value is generated according to what the environment can compute according to the environment cryptographic reductions of Section 5.2.1 ($\vdash_{env} K \rightsquigarrow K' \triangleright v$).

Rule RILC-Sem-Other relies on the previously defined semantics to execute a reduction within the reactive program (note that this rule can only be triggered when the program has the write token).

Finally, Rule RILC-Sem-Error halts the computation whenever the *error* expression is executed.

$$\frac{}{\Omega \overset{\epsilon}{\Longrightarrow} \Omega}\ \text{(Trace-Refl)} \qquad \frac{\Omega \xrightarrow{\tau} \Omega'' \quad \Omega'' \overset{\overline{\tau}}{\Longrightarrow} \Omega'}{\Omega \overset{\tau\cdot\overline{\tau}}{\Longrightarrow} \Omega'}\ \text{(Trace-Action)} \qquad \frac{\Omega \xrightarrow{\epsilon} \Omega'' \quad \Omega'' \overset{\overline{\tau}}{\Longrightarrow} \Omega'}{\Omega \overset{\overline{\tau}}{\Longrightarrow} \Omega'}\ \text{(Trace-No-Action)}$$

Reactive Program Traces. The following judgements define when does a reactive program generate a (set of) traces.

$W; \lambda; n^{rnd} \rightsquigarrow \{\overline{t}\}$	Reactive program W produces a set of traces $\{\overline{t}\}$ with security parameter λ and randomness n^{rnd}
$Behav(W, \lambda) = \{\overline{t}\}$	Behaviours (i.e., set of traces $\overline{t}$) of a reactive program W with a security parameter λ

Before we explain each judgement and their rules, we describe some auxiliary functions.

$$\mathsf{NR}(\lambda) = \left\{ n \;\middle|\; n : [Bits] \text{ and } \|[Bits]\| = f(\lambda) \text{ and } f : \mathbb{N} \to \mathbb{N} \text{ and } \vdash \mathtt{poly}(f) \right\}$$

$\mathsf{NR}(\lambda)$ generates all the random bitstrings $\{n\}$ that a reactive program can use, and the length of these bitstrings is a polynomial function ($\vdash \mathtt{poly}(f)$) of the security parameter λ. In the following, we use $\|\cdot\|$ to indicate the length of a list, or the cardinality of a set.

$$\frac{W = D; \Pi; I; X \qquad \vdash W \qquad \Xi = I \cup X \qquad \Sigma = \mathsf{dom}(\Pi) \cup \mathsf{dom}(\Xi)}{\Omega_0\left(W; \lambda; n^{rnd}\right) = \left\langle \varpi; K_0(\lambda, n^{rnd}); \Xi; \Sigma; \Pi \right\rangle}\ \text{(RILC-Initial State)} \qquad \frac{\nexists \Omega', \tau.\ \Omega \overset{\tau}{\hookrightarrow} \Omega'}{\vdash \Omega : term}\ \text{(RILC-Terminal State)}$$

Given a complete (as per the definition of "complete" given in Section 2.2) reactive program W (and the security parameter, and a randomness bitstring n^{rnd}), Rule RILC-Initial State defines the starting state for that program. Dually, Rule RILC-Terminal State tells that a configuration is terminal, i.e., it cannot step any more.

$$W; \lambda; n^{rnd} \rightsquigarrow \left\{ (\overline{\tau}, \rho) \;\middle|\; \exists \Omega.\ \Omega_0\left(W; \lambda; n^{rnd}\right) \overset{\overline{\tau}}{\Longrightarrow} \Omega \text{ and } \vdash \Omega : term \text{ and } \rho = 1/(\|\mathsf{NR}(\lambda)\|) \right\}$$

(RILC-Interaction Traces)

Rule RILC-Interaction Traces calculates the set of traces of a complete reactive program. The probability of each trace is calculated as the probability of obtaining the trace over all randomnesses that exist.

$$Behav(W, \lambda) = \left\{ (\overline{\tau}, \rho) \;\middle|\; W; \lambda; n^{rnd} \rightsquigarrow (\overline{\tau}, \rho_0) \text{ and } \rho = \textstyle\sum_{\rho_0} \text{ and } n^{rnd} \in \mathsf{NR}(\lambda) \right\}$$

(RILC-Interaction Traces)

Rule RILC-Interaction Traces calculates the behaviour of a complete reactive program. The probability ρ of each trace is calculated as the sum of all the probabilities ρ_0 to generate that trace calculated with any possible randomness n^{rnd} taken from the space of randomnesses $\mathsf{NR}(\lambda)$.

5.3 Properties of RILC

RILC enjoys a number of properties, first that its additions from Section 5.2 still preserve well-typedness and confluence from the original ILC work (Section 5.3.1). Then, RILC fulfils the axioms of Section 4, so it can be used in our connection and it enjoys the composition theorem and the dummy adversary one (Section 5.3.2).

5.3.1 Language Properties. In order to state confluence for RILC we define two projection functions on traces that extract the inputs $(\cdot|_I)$ and outputs $(\cdot|_O)$ actions of a trace:

$$\frac{}{\varnothing|_I = \epsilon}\ \text{(Inputs - empty)} \qquad \frac{\tau = \text{send } v \text{ on } c?}{\tau \cdot \overline{\tau}|_I = \tau \cdot \overline{\tau}|_I}\ \text{(Inputs - ?)} \qquad \frac{\tau = \text{send } v \text{ on } c!}{\tau \cdot \overline{\tau}|_I = \overline{\tau}|_I}\ \text{(Inputs - !)}$$

$$\frac{}{\varnothing|_O = \epsilon}\ \text{(Optputs - empty)} \qquad \frac{\tau = \text{send } v \text{ on } c?}{\tau \cdot \overline{\tau}|_O = \overline{\tau}|_O}\ \text{(Outputs - ?)} \qquad \frac{\tau = \text{send } v \text{ on } c!}{\tau \cdot \overline{\tau}|_O = \tau \cdot \overline{\tau}|_O}\ \text{(Outputs - !)}$$

Additionally, we formalise the fact that during two executions the environment changes the cryptographic state in the same manner. We say that two such executions are consistent wrt cryptographic inputs, denote this fact as $\vdash \cdot \,||\, \cdot : CIC$ and formalise it as follows:

$$\frac{}{\vdash \Omega_1 \overset{\epsilon}{\Longrightarrow} \Omega_1 \,||\, \Omega_2 \overset{\epsilon}{\Longrightarrow} \Omega_2 : CIC}\ \text{(Cryptographic input Consistency - Base)}$$

$$\frac{\vdash \Omega_1'' \overset{\overline{t}}{\Longrightarrow} \Omega_1' \,||\, \Omega_2'' \overset{\overline{t}}{\Longrightarrow} \Omega_2' : CIC \qquad \vdash \Omega_1 \xrightarrow{\tau} \Omega_1' \,||\, \Omega_2 \overset{\tau}{\Longrightarrow} \Omega_2'' : CIC}{\vdash \Omega_1 \xrightarrow{\tau} \Omega_1'' \overset{\overline{t}}{\Longrightarrow} \Omega_1' \,||\, \Omega_2 \xrightarrow{\tau} \Omega_2'' \overset{\overline{t}}{\Longrightarrow} \Omega_2' : CIC}\ \text{(Cryptographic input Consistency - Inductive)}$$

$$\frac{\tau = \epsilon \vee \tau = \tau!}{\vdash \Omega_1 \xrightarrow{\tau} \Omega_1' \,||\, \Omega_2 \xrightarrow{\tau} \Omega_2' : CIC}\ \text{(Cryptographic input Consistency - Single-nop)} \qquad \frac{\Omega_1'.K = \Omega_2'.K}{\vdash \Omega_1 \xrightarrow{\tau?} \Omega_1' \,||\, \Omega_2 \xrightarrow{\tau?} \Omega_2' : CIC}\ \text{(Cryptographic input Consistency - Single-in)}$$

RILC has a full confluence property (Theorem 9) telling that if a program state Ω performs two different traces up to two terminal states Ω_1 and Ω_2 with the same environment-supplied inputs, and with the same changes to the cryptographic state from the environment, then the terminal states are renamings of each other (with renaming function $g(\cdot)$), and the outputs of the traces are also the same.

Theorem 9 (Full Confluence)**.**

$$\text{if } \Omega \overset{\overline{t}}{\Longrightarrow} \Omega_1 \text{ and } \Omega \overset{\overline{t}'}{\Longrightarrow} \Omega_2 \text{ and } \vdash \Omega_1 : term \text{ and } \vdash \Omega_2 : term$$

$$\text{and } \overline{t}|_I = \overline{t}'|_I \text{ and } \vdash \Omega \overset{\overline{t}}{\Longrightarrow} \Omega_1 \,||\, \Omega \overset{\overline{t}'}{\Longrightarrow} \Omega_2 : CIC$$

$$\text{then } \Omega_1 = g(\Omega_2) \text{ and } \vdash g(\Omega_2) : term \text{ and } \overline{t}|_O = \overline{t}'|_O$$

The proof of Theorem 9 is an unsurprising induction over the generated traces, with the silent actions case being inherited from Liao et al. [73]. The only interesting case is the single visible action lemma (Lemma 1).

Lemma 1 (Generalised Confluence Single)**.**

$$\text{if } \Omega \xrightarrow{\tau} \Omega_1 \text{ and } f(\Omega) \xrightarrow{\tau'} \Omega_2 \text{ and } \tau|_I = \tau'|_I \text{ and } \Omega_1.K = \Omega_2.K$$

$$\text{then } \Omega_1 = g(\Omega_2) \text{ and } \tau|_O = \tau'|_O$$

In the proof of Lemma 1, we have to reason about the rules generating the single actions, which are the key additions to RILC. Thus, this proof essentially shows that our additions to RILC still keep the language confluent, assuming that the environment provides the same inputs and the same changes to the cryptographic state.

5.3.2 Fulfilling the Axioms of Section 4. The Axioms of Section 4 are split in two groups, those needed for the composition theorem and those needed for the dummy adversary one.

Axioms for Composition. In order to state the Axioms of Section 4.1 in RILC, we need to instantiate the different abstract composition operators ($\bowtie$, $\circledast_{\mathrm{T}}^{S}$, $\oplus_{\mathrm{T}}^{S} \circledcirc_{\mathrm{T}}^{S}$) in RILC. Since RILC has a single composition operator ($\bowtie$ from Rule RILC-Linking), we instantiate all the operators with $\bowtie$. We believe in other languages where the distinction between attacker and program needs to be carried around (e.g., as in the work of El-Korashy et al. [51]), we could not instantiate all the operators with a single one, but we would have to define different ones.

The key property that we rely on for proving the axioms is linking commutativity (Lemma 2), whose proof is straightforward and thus left in the technical report [86].

Lemma 2 (Linking Commutativity)**.** $A \bowtie M \eqsim M \bowtie A$

Below are the statements of the axioms instantiated for RILC. As their proofs are tedious and not insightful, we leave them in the technical report [86]. First we present the ones related to composition:

Axiom 5 $\forall A_0, M.\ \exists A_1, A_2.\ A_0 \bowtie M \eqsim (A_1 \bowtie A_2) \bowtie M$
Axiom 6 $(A_1 \bowtie A_2) \bowtie (M1 \bowtie M2) \eqsim (A_1 \bowtie M1) \bowtie (A_2 \bowtie M)$
Axiom 7 if $M1 \eqsim M2$ then $M3 \bowtie M1 \eqsim M3 \bowtie M2$

From these Axioms, we can derive Corollary 1 below, which is Theorem 6 (Composition in *RC*) specified for RILC:

Corollary 1 (Composition for RILC)**.**

$$\text{if } \forall A.\exists A'.\ A \bowtie [\![P]\!] \eqsim A' \bowtie P \text{ then } \forall A''.\exists S.\ A'' \bowtie P_s \bowtie [\![P]\!] \eqsim S \bowtie P_s \bowtie P$$

Axioms for the Dummy Attacker. The axioms related to the dummy attacker (Section 4.2) cannot possibly be stated (let alone proved) in a general fashion. In fact, different modules (which model different protocols) provide different signatures, and thus there is not really a single dummy attacker that fits all protocols.[9] However, we notice that once the signature of a module is fixed, and the meaning of each imported and exported channels is known, it is possible to devise a dummy attacker for that protocol. Even more, we conjecture that for every protocol it is possible to devise a dummy attacker (though we leave stating and proving such a conjecture for future work). Thus, we do not provide a general proof of Axiom 8 and Axiom 9, but leave them to be proved on a per-protocol basis.

Specifically, we will report on the proofs of those Axioms for the specific protocol we consider in this paper in Section 6.2.4.

With our candidate formal language for instantiating the connection, we now turn to reaping its benefits and show how to derive UC proofs as *RC* ones.

6 RIGOROUS, SCALABLE UC PROOFS AS *RC* PROOFS

This section first provides some background on UC and *RC* proofs and it clarifies how to use our connection to provide UC proofs via *RC* ones (Section 6.1). Then it provides two instances of UC proofs done via *RC* ones. The first one is for a known result: UC of a single-bit commitment in the static corruption setting [41, 73] (Section 6.2). The second one is instead for a simple but (to our knowledge) novel result: UC of the same protocol, but with adaptive corruptions (Section 6.3).

[9] We note that there may be some simplifying assumptions that fix the channels in a signature (e.g., one channel per protocol with a unit that is responsible to dispatch the messages to the protocol parties), but that is beyond the scope of this paper.

6.1 Background on UC and *RC* Proofs

We now recount UC (Section 6.1.1) and *RC* proofs (Section 6.1.2), before drawing conclusions on the benefits to reap from each approach (Section 6.1.3).

6.1.1 UC *Proofs.* As mentioned earlier, the vast majority of UC proofs exploits the dummy-adversary theorem (Theorem 7) and thus consist of two steps. First, authors establish the existence of the simulator by stating it explicitly. There is usually little explanation about why it is defined as it is, since the simulator is the result of a fair amount of trial and error. Second, the authors prove the indistinguishability property at the heart of UC: $\textsc{Exec}\,(\mathsf{Z}, \mathsf{A}_\mathsf{d}, \pi) \approx \textsc{Exec}\,(\mathsf{Z}, \mathsf{S}, \mathsf{F})$. This is usually an induction over the length of the trace from the perspective of the environment, or more precisely, over the number of outgoing messages from the environment, which mark the beginning of what is often called an *epoch*. The epoch ends when the environment regains control by receiving a message from the attacker or protocol. For each epoch, these proofs show (via case distinction over the message) that some state invariant holds; that will ensure that the message the environment receives at the end of the epoch is indistinguishable in the ideal and the real world. Consequently, the state invariant links the state of the parties in the ideal world (S, F) and in the real world ($\mathsf{A}_\mathsf{d}, \pi$) to each other. A recurring pattern in UC proofs is that the simulator S tracks a large part of the state of the protocol π to correctly simulate behaviour that is independent from F.

The main work of the proof is hence in the aforementioned case distinction: no matter what the environment sends to the protocol/ideal functionality, the state invariant holds. The majority of these cases is about the simulator correctly tracking the protocol state. For one, this is tedious: state invariants can become very large, since interactions in the ideal world may involve multiple messages between S and F. For two, and somewhat tragically, this is mechanical but not easily mechanisable (as admitted by the authors of one mechanised UC proof [43], see Section 6.1). The mathematician is essentially performing a symbolic execution in a deterministic system. If the language used to describe protocol, simulator and functionality were to have a formal semantics, this tedious task could be automated. In many pen-and-paper proofs, this language has no formal semantics and it is thus not very surprising, albeit disappointing, that these laborious steps are skipped entirely. The proof concentrates on the few cases where a cryptographic argument is necessary, establishing a reduction of this form: if, given the state invariant, the environment can compute a message, then the same environment can be transformed into an attacker for some cryptographic game

Summarizing, there is a lot of potential for automation in these proofs, should they be conducted in a formal language. We believe some of the low-hanging fruits here are (i) the symbolic execution to prove state equivalence in the majority of cases and (ii) the automated synthesis of the state equivalence relation. A more challenging task is the partial synthesis of the simulator. Given a partial simulator and a partial state equivalence that fixes the cryptographically interesting parts of the interaction, provide a simulator and state equivalence relation that produces equivalent traces for the interactions that are not defined by the partial simulator.

While the informal description above captures the majority of UC proofs, we note that some follow a more structured approach. The game-playing technique [31] consists of structuring cryptographic proofs as a series of small syntactic refinement steps in an informal pseudocode language. The technique is also sometimes used in UC proofs (e.g., [16, 42]) and it has been mechanised in EasyUC but, quoting Canetti et al. [43]: "despite the relative simplicity [of their case studies, proving UC] took an immense amount of work" [43]. The authors point out the need for a domain specific language that is coroutine-based, as opposed to the underlying EasyCrypt [28] language they used, as well as the lack of symbolic evaluation. Moreover, proving composition inside EasyCrypt was only possible for specific instances, but not in general. The authors state that

a general proof must be performed at the level of EasyCrypt's metatheory, i.e., without automation. Our composition axiom can help this task. Later work in EasyCrypt [27], heavily improved on the proof effort and automation by aligning the message passing semantics more closely to EasyCrypt's procedure-based communication model. As this enforces a hierarchical communication model (in contrast to UC, where the environment can directly access lower level functionalities), the authors suggest to think of their work as a backend for [43] and leave a sound translation for future work. The main concern of their work [27] is mechanising the complexity analysis of programs – which is a precondition for the validity of composition in the computational setting.

6.1.2 RC Proofs. Most *RC* proofs follow a proof technique inherited from secure compilation works [80] called *Backtranslation* [11, 12, 14, 83, 85]. The goal of a backtranslation is to define how to construct any source attacker (the existentially-quantified A in Definition 4) starting from whatever target information available (i.e., what preceeds said existentially-quantified A). If the used information is the target attacker $\mathtt{A}$, then the proof is called *Context-based Backtranslation*, if the used information is the target trace, then it is called *Trace-based Backtranslation*.

Once the backtranslation is defined, the *RC* proofs essentially proceed by induction over the target trace that needs to be replicated in the source. Most times those traces follow this inductive principle: they are either empty or the concatenation of a trace with a list of attacker-generated actions ending with a control-transferring action to the program ($\overline{\tau} \cdot \tau?$), followed by a sequence of program-generated actions ending with a control-transferring action to the context ($\overline{\tau}' \cdot \tau!$).

The proof sets up a simulation argument between the target context performing $\overline{\tau} \cdot \tau?$ and its backtranslated counterpart, which needs to be proven to produce $\overline{\tau} \cdot \tau?$ in the source. Following established compilation-style proofs [72], this is done by setting up a cross-language state relation ($\Omega \sim \Omega$) between the states of the source attacker (Ω) and those of its backtranslation-generated counterpart (Ω). Then, after the ?-decorated action, control is transferred to the compiled program. The proof then proceeds by a classical compiler correctness argument which ensures that the program performs $\overline{\tau}' \cdot \tau!$ given that its compilation perform $\overline{\tau}' \cdot \tau!$. This proof is carried out with a different cross-language state relation relating states of the source program and its compilation ($\Omega \approx \Omega$). Notably, the relation when control is in the attacker code ($\sim$) tends to enforce a stronger invariant than the other one ($\approx$), and thus there is need to weaken and strengthen the relation. The weakening and strengthening often happens when the compiler performs some security-related checks that are placed as a wrapper around the program. It is in fact common for secure compilers to act as "security wrappers" around correctly-compiled code [6, 34, 49, 82].

RC works that use the backtranslation proof technique typically do not use reactive languages, and thus they cannot write the analogous of the dummy attacker theorem. Thus, backtranslation proofs require quite some effort, with research being active in simplifying said effort [51]. This effort is especially great when these proofs are mechanised. At the time of writing, the only such mechanised effort required 20K lines of Coq just for a part of the backtranslation correctness proof [50].

6.1.3 The Best of Both Worlds. In the following, we will use our connection (Theorem 3) to provide rigorous UC proofs, relying on the benefits of *RC* proofs. Specifically, we will consider a protocol and its ideal functionality, coded in RILC. By using the RILC semantics, we will perform the reductions in the real world (for now manually) and compute the real-world traces. Then, we will perform the ideal world reductions and show that the ideal traces they yield are the same (trace and probability-wise) as the real-world ones. This will let us conclude that the compiler from the ideal functionality to the protocol is *RHP*, and thus the latter UC-realises the former.

6.2 UC Commiments as *RC* Proofs for the Static Corruption Model

In order to perform this proof we need to define the following RILC processes:

(1) the ideal functionality F (Section 6.2.1);
(2) the protocol P, Q that UC-realises F (Section 6.2.2);
(3) the simulator S used in the proof (Section 6.2.3).

Finally, the proper proof is presented in Section 6.2.4, followed by a discussion Section 6.2.5. This proof rests on a few assumptions, which we inherit from the original UC proofs from Canetti and Fischlin [41], and that we present below.

Assumptions for the Static Corruption Commitment. Below are the assumptions that hold for the protocol of this section. Note that these assumptions are not imposed by us, but they are inherited by the work that defined the protocol in the first place. We simply recap them here for simplicity.

(1) the protocol assumes authenticated channels, i.e., if a party A receives a message m, it knows the identity of the party B that sent m.
(2) the only party that can be corrupted is the committer (P), since corrupting the receiver does not let the attacker break any of the commitment properties. This is borrowed from the simulator of Canetti and Fischlin [41], which only considers the effect of corruption on commitments created by the corrupted party—in other words, corruption is only relevant for a party when that party acts as a sender.

6.2.1 Ideal Functionality. The ideal functionality is presented in Listing 1, with comments explaining the code. In all our code snippets when we write a channel XY, it is a channel used from process X to send a message read by process Y.

```
process F =
  rd Commit b from Read(PF)      // receive a commitment message for a bit 'b'
  wr Receipt to Write(FS)        // acknowledge that the commitment has been initiated
  rd Open from Read(PF)          // receive a message to open the commitment
  wr Opened b to Write(FS)       // open the commitment to the right bit 'b'
```

Listing 1. Ideal Functionality for the single-bit commitment.

An interesting bit of this code is its *module signature*, which is in Listing 2. The reason we deem such signatures interesting is that it describes the channels that will be used.

```
imports If = Write(FS)
definitions Df = Read(PF), Write(FS)
exports Xf = Read(PF)
```

Listing 2. Module signature for ideal functionality for the single-bit commitment.

6.2.2 Protocol. The protocol consists of two parties (a committer P and a receiver Q) whose code is in Listing 3 and of a common reference string whose code is in Listing 4. In the static corruption model, corruption is defined at the beginning of the computation with a message from the environment. That is what happens on Line 2 and that is the reason why the code of the committer P is split in two parts: the non-corrupted behaviour (Line 3 to Line 20) and the corrupted one (Line 22 to Line 30). Both parts contain effectively the same preamble (Line 4 to Line 9). Those are additional messages that we inserted *before* the proper protocol to fix the common reference string (CRS)–related interleavings, since this simplifies the proof. The CRS (presented in Listing 4 and described below) is a piece of code that P, Q, and Z all interact with. To both simplify the proof

```
process P =
  rd M from Read(ZP)                    // receive the static corruption message
  if M == CrptNO then                   // in case no party is corrupted
    wr StartCrs to Write(PCrs)          // start the CRS functionality
    rd Started from Read(CrsP)          // ack the CRS started
    wr Ok to Write(PZ)                  // return control to the environment,
                                        //  which will call the CRS
    rd WaitCrs from Read(CrsP)          // synchronise with the environment
    wr Waited to Write (PCrs)
                                        // here starts the commitment protocol,
                                        //  as taken from the literature
    rd Commit b from Read(ZP)           // receive a commitment message for a bit 'b'
    wr GetCRS to Write(PCrs)            // request the CRS
    rd PublicStrings s pk0 pk1 from Read(CrsP)      // receive the CRS
    let r = takernd
    let y = (if b == 0 then prg pk0 r else xors(prg(pk1, r), s))  // set y to
                                        //  the commitment of bit 'b' using the 'prg'
    wr Commit' y to Write(PQ)           // send the commitment 'y' to the receiver
    rd Open from Read(ZP)               // receive a message to open the commitment
    wr Open' b r to Write(PQ)           // forward that request to the receiver
  else                                  // if the corruption message is not 'CrptNO',
                                        //  the committer is corrupted
    wr StartCrs to Write(PCrs)          // these synchronisation messages are as above
    rd Started from Read(CrsP)
    wr Ok to Write(PZ)
    rd WaitCrs from Read(CrsP)
    wr Waited to Write (PCrs)
                                        // below is the corrupted committer: a proxy
                                        //  from the environment to the receiver
    loop(2)(fwd(Read(ZP))(Write(PQ)))   // we expect 2 iterations, thus the bound

process Q =
  rd Commit' y from Read(PQ)      // receive the commitment 'y' from the committer
  wr GetCRS to Write(QCrs)        // request the CRS
  rd PublicStrings s pk0 pk1 from Read(CrsQ)    // receive the CRS
  wr Receipt to Write(QZ)         // acknowledge the commitment has been initiated
  rd Open' b r from Read(PQ)      // receive original commitment bit 'b'
                                  //  and randomness 'r'
                                  // below: calculate if the communication
                                  //  of 'b', 'r' and 'y' was done properly
  if (b == 0 && y == prg(pk0, r)) \/ (b == 1 && y == xors(prg(pk1, r), s))
    wr Opened b to Write(QZ)
  else error                      // some parameter does not match: abort
```

Listing 3. Single-bit commitment protocol.

and to establish a security argument (as explained in Section 6.2.4), we fix the interaction to be first Z, then P, then Q.

Rather than describing the exchange in text, we provide a diagrammatic representation of the expected message exchanges in Figure 5. Apart from the additional code that explicitates the corruption and from the code that simplifies the rigorous reasoning in the proofs, the essence of the protocol code is the same as in its previous definition by Canetti and Fischlin [41].

Intuitively, the CRS functionality lets the parties read from a bit string that is sampled from a specified distribution during a one-time setup phase. For the purposes of the protocol, this setup phase is assumed to be run honestly. The protocol's use of a CRS sidesteps the impossibility of constructing a simulator for secure commitments in the *plain model* (a setting that makes no assumptions about an honest setup phase); this impossibility result was proved by Canetti and

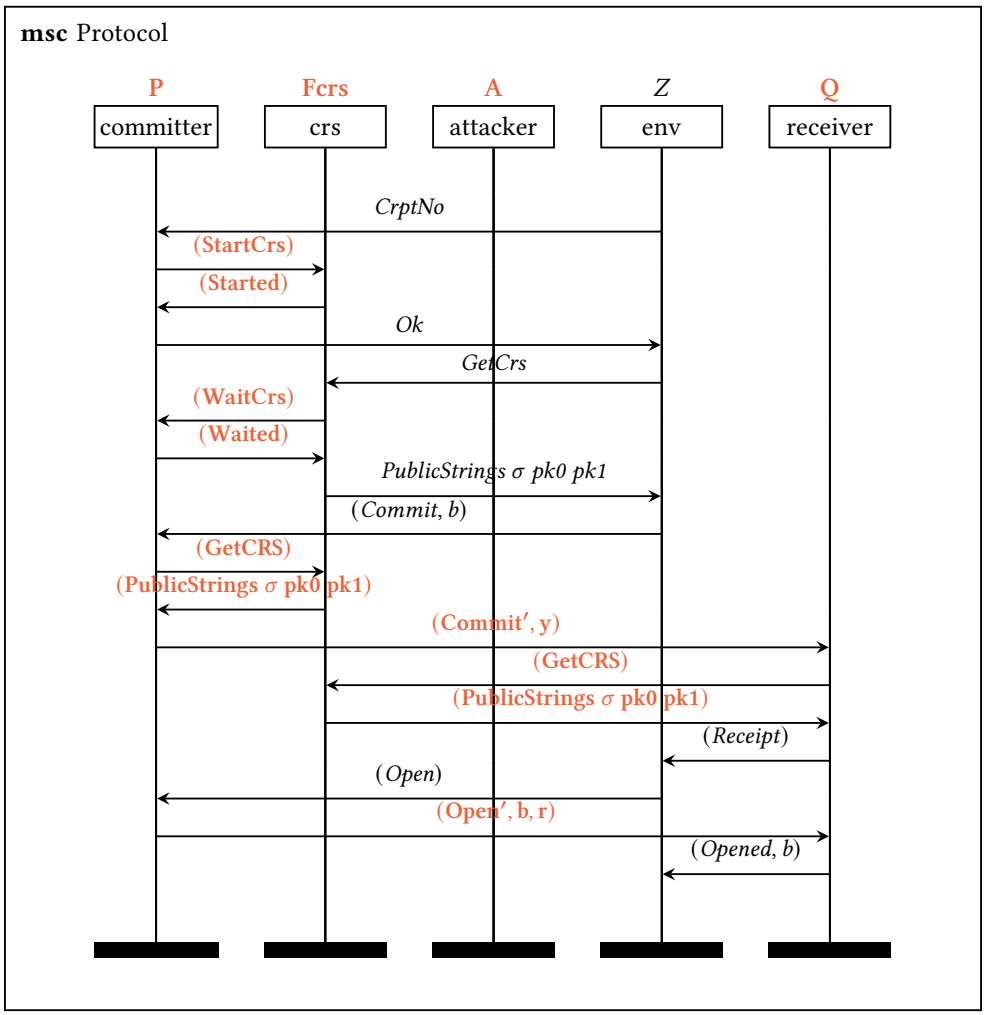

Fig. 5. Diagram representation of the protocol interaction in the no-corruption case. Red actions are *internal* steps while *black* ones are those that constitute a trace.

Fischlin [41, §3]. As for the protocol, the core intuition behind this code and its original formulation is unchanged from the original paper [41].

The signature for the protocol (in Listing 5) highlights an interesting difference with the signature of the ideal functionality presented in Listing 2. In fact, this signature is *different* from the one of the ideal functionality: *and rightly so*! In fact, from a module perspective, at the protocol level, this code is all we consider, while at the functionality level, we do not just consider F, but also the simulator (which we discuss next). Thus, the signature of the protocol module should be the same as the signature of the module obtained from linking the functionality and the simulator. As we show below, this is the case, but in order to argue this point, we need to present the simulator first.

6.2.3 Simulator. The simulator for this proof contains a number of code snippets: the simulator, the fake common reference string, and the simulated parties. The simulator is presented in Listing 6.

```
process Crs =
  rd StartCrs from Read(PCrs)
  wr Started to Write(CrsP)
  rd GetCrs from Read(ZCrs)
  let s = takernd
  let r0 = takernd
  let r1 = takernd
  let pk0, _ = keygen(r0)
  let pk1, _ = keygen(r1)
  wr WaitCrs to Write(CrsP)
  rd Waited from Read(PCrs)
  wr PublicStrings s pk0 pk1 to Write(CrsZ)
  rd Getcrs from Read(PCrs)
  wr PublicStrings s pk0 pk1 to Write(CrsP)
  rd GetCrs from Read(QCrs)
  wr PublicStrings s pk0 pk1 to Write(CrsQ)
```

Listing 4. Protocol CRS

```
imports Ip = Write(PCrs), Write(PZ), Write(PQ)
definitions Dp = Write(PCrs), Write(PZ), Write(PQ), Read(ZP), Read(CrsP)
exports Xp = Read(ZP), Read(CrsP)
imports Iq = Write(QCrs), Write(QZ)
definitions Dq = Write(QCrs), Write(QZ), Read(PQ), Read(CrsQ)
exports Xq = Read(PQ), Read(CrsQ)
imports Icrs = Write(CrsP), Write(CrsZ), Write(CrsQ)
definitions Dcrs = Write(CrsP), Write(CrsZ), Write(CrsQ), Read(PCrs), Read(ZCrs), Read(
    QCrs)
exports Xcrs = Read(PCrs), Read(ZCrs), Read(QCrs)

// the resulting signature after linking P, Q, and Crs is
imports I = Write(PZ), Write(QZ), Write(CrsZ)
definitions D = Dp ∪ Dq ∪ Dcrs
exports X = Read(ZP), Read(ZCrs)
```

Listing 5. Protocol Interface

The simulator receives the static corruption message (Line 2) and behaves differently in case P is corrupted (Line 12) or not (Line 31). In either case, the simulator starts the fake crs and receives the fake parameters. If P is not corrupted, the simulator acts as a proxy for the messages it receives Lines 8 and 11. If P is corrupted, the simulator receives the intermediate $Commmit'\ y'$ and $Open'$ messages (Line 18 and Line 32 respectively). Then, it uses its knowledge of the fake CRS to calculate whether the received commitment bit y' is in the image of the 0 or 1 key. Once it learns this information, it sends the correct bit (0 or 1) to the ideal functionality (Line 26 and Line 28 respectively). From here on, the simulator acts as a proxy, and in the end it checks whether the opened bit was correct (Line 33).

The fake common reference string is responsible for setting up a common reference string whose backdoor is known to the simulator (Listing 7). The fake CRS is a process we inherit from the original proof of UC for this protocol [41]. The reason we need this in the ideal world is dictated by some reductions that happen in the real world, which we now discuss. Essentially, in the case **P** is corrupted, it could behave maliciously or honestly. In order to detect the honest case, the simulator must know whether its messages are conformant to the data communicated from the CRS (i.e., whether it used the string **s**, and the keys **pk0** and **pk1**). Thus, the first thing the simulator does is starting the fake CRS process, which computes a known common reference string s and that communicates the string as well as both the keys and the trapdoors to the simulator (Line 8). With

```
process S =
  rd !M from Read(ZP)
  if M == CrptNO then
    wr CrptNO to Write(SP)
    rd !Ack from Read(PS)
    wr FakeSetup to Write(SCrs)                 // starts the fake CRS
    rd !Fake pk0 pk1 td0 td1 s from Read(CrsS)  // receives the fake CRS parameters
    wr Ok to Write(SP)
    rd !Receipt from Read(FS)
    wr Receipt to Write(SQ)
    fwd(Read(FS))(Write(SQ))
  else
    wr CrptP to Write(SP)
    rd !Ack from Read(PS)
    wr FakeSetup to Write(SCrs)                   // starts the fake CRS
    rd !Fake pk0 pk1 td0 td1 s from Read(CrsS)    // receives the fake CRS parameters
    wr Ok to Write(SP)
    rd !Commit' y from Read(PS)                 // receive the Commit' message from P
    let g0 = (invert(⟨pk0, td0⟩, y))            // use the trapdoor to calculate if the
                                                //  committed bit 'y'' in the image of
                                                //  the key for 0
    let g1 = (invert(⟨pk1, td1⟩, xors(y, s)))   // use the trapdoor to calculate if the
                                                //  committed bit 'y'' in the image of
                                                //  the key for 1
    if g0 then
      wr Commit 0 to Write(SP)    // instruct P (and then the F) to commit to 0
    elif g1 then
      wr Commit 1 to Write(SP)    // instruct P (and then the F) to commit to 1
    else
      wr Commit 0 to Write(SP)    // fake commitment, just to resume other processes
    fwd(Read(FS)(Write(SQ)))
    rd !Open' b r from Read(PS)   // receive the Open' message from a (malicious?) P
    if (b == 0 && g0) \/ (b == 1 && g1) then
      wr Open to Write SP         // instruct P (and then the F) to Open
      fwd(Read(FS))(Write(SQ))    // forward the Opened message to Q
    else error
```

Listing 6. Simulator for the single-bit commitment.

```
process FAKECRS =
  rd !FakeSetup from Read(SCrs)
  let pk0, td0 = keygen([1..1])   // lambda-long ones
  let pk1, td1 = keygen([1..1])   // lambda-long ones
  let r0 = takernd
  let r1 = takernd
  let s = xors(prg(pk0, r0), prg(pk1, r1))
  wr Fake pk0 pk1 td0 td1 s to Write(CrsS)
  rd GetCrs from Read(ZCrs)
  wr WaitCrs to Write(CrsP)
  rd Waited from Read(PCrs)
  wr PublicStrings s pk0 pk1 to Write(CrsZ)
```

Listing 7. Fake CRS for the single bit commitment.

the trapdoor, the simulator will be able to undestand if the message received from P (Line 18) is honest or not.

The dummy parties are elements that are present in the literature too, and here we prove that they are needed in order to make the signatures of the modules match (Listing 8). Essentially, the

```
process P =
  rd !M from Read(ZP)
  wr M to Write(PS)
  rd !M from Read(SP)
  if M == CrptNO then
    wr Ack to Write(PS)
    rd Ok from Read(SP)
    wr Ok to Write(PZ)
    rd WaitCrs from Read(CrsP)
    wr Waited to Write (PCrs)
    loop(2)(
      fwd(Read(ZP))(Write(PF))
    )
  else // M == CrptP
    wr Ack to Write(PS)
    rd WaitCrs from Read(CrsP)
    wr Waited to Write (PCrs)
    loop(2)(
      fwd(Read(ZP))(Write(PS))
      fwd(Read(SP))(Write(PF))
    )

process Q =
  fwd(Read(SQ))(Write(QZ))
```

Listing 8. Simulator Parties

dummy parties act as proxies: they forward environment messages to the functionality (Lines 3 and 19), and simulator messages to the environment (Lines 8 and 24) (or to the functionality Line 12).

The signature of the simulator is presented in Listing 9.

```
imports Is = Write(SP), Write(SCrs), Write(SQ)
definitions Ds = Write(SP), Write(SCrs), Write(SQ), Read(ZP), Read(PS), Read(CrsS), Read
    (FS)
exports Xs = Read(ZP), Read(PS), Read(CrsS), Read(FS)
imports Ifake = Write(CrsS), Write(CrsP), Write(CrsZ)
definitions Dfake = Write(CrsS), Write(CrsP), Write(CrsZ), Read(SCrs), Read(ZCrs), Read(
    PCrs)
exports Ifake = Read(SCrs), Read(ZCrs), Read(PCrs)
imports Idp = Write(PS), Write(PZ), Write(PCrs), Write(PF)
definitions Ddp = Write(PS), Write(PZ), Write(PCrs), Write(PF), Read(ZP), Read(SP), Read
    (CrsP)
exports Idp = Read(ZP), Read(SP), Read(CrsP)
imports Idq = Write(QZ)
definitions Ddq = Write(QZ), Read(SQ)
exports Idq = Read(SQ)

// the resulting signature after linking P, Q, FAKECRS, and S is
imports I = Write(CrsZ), Write(PZ), Write(PF), Write(QZ)
definitions D = Ds ∪ Dfake ∪ Ddp ∪ Ddq
exports X = Read(ZP), Read(FS), Read(ZCrs), Read(ZP)


// the resulting signature after linking the result above with F is
imports I = Write(CrsZ), Write(PZ), Write(QZ)
definitions D = Ds ∪ Dfake ∪ Ddp ∪ Ddq ∪ Df
exports X = Read(ZP), Read(ZCrs)
```

Listing 9. Simulator Signature.

When linking the simulator with the ideal functionality, we obtain a single process whose signature – in terms of imports and export – is the same of the protocol signature from Listing 5. This is necessary, since such a (static) difference would already make any UC (or *RHP*) result impossible.

6.2.4 RC Proof. We can now define a compiler that translates Listing 1 into Listing 3 (Definition 12) and prove it is *RHP* (Theorem 10).

Definition 12 (Compiler for the Static Corruption Commitment)**.**

$$[\![\cdot]\!] \stackrel{\text{def}}{=} [\![P_{comm}]\!] \rightharpoonup \mathbf{P}_{\mathbf{comm}} \qquad \text{where } P_{comm} \text{ is Listing 1}, \ \mathbf{P}_{\mathbf{comm}} \text{ is Listing 3}$$

Theorem 10 (The $[\![\cdot]\!]$ Compiler is *RHP*)**.**

$$\vdash [\![\cdot]\!] : RHP$$

Like most proofs, the one of Theorem 10 is tedious, so we now report the main insights we gained while doing it and leave the full proof in the technical report [86]. We believe these steps may be helpful for those that want to have a paper proof before attempting a mechanisation such as the one presented in the next section (as we did).

General structure: First, we devise a dummy attacker for the commitment and prove Axiom 9 and Axiom 8, which yield Theorem 8 (Dummy Attacker in *RC*) (again, these uninformative proofs are left in the technical report). This lets us get rid of the target attacker and, in turn, ensures that we have all the code (protocol, ideal functionality, simulator) whose behaviour needs to be analysed for the proof.

Then, the proof proceeds by specialising the trace-based backtranslation proof technique used in secure compilation work [11, 79, 83]. Intuitively, the proof consists of analysing the target-level traces and generate the simulator in such a way that it replicates all ?-decorated actions in the trace. The !-decorated actions are then generated by the source program, provided it is called with the correct arguments. Ensuring the source program is called with the correct arguments can be complex, for example the proofs for the one-bit commitment presented here rely on a *FAKECRS* functionality and on the invertibility of the *prg* function. Since the application of the backtranslation proof technique is more interesting in the adaptive corruption case, we provide more details there (see Section 6.3.4).

Once the simulator is devised, the proof proceeds by matching the reductions in the source (P_{comm} plus simulator) with those in the target ($\mathbf{P}_{\mathbf{comm}}$), while keeping track of what actions generate traces. While seemingly dull, this exercise alone is useful in getting rid of inconsistencies such as the ideal functionality sending a message with some data, and the protocol sending a message with more data.

Interleavings: One complication when doing this proof is the amount of interleavings that arise due to the choices that the environment can do. For example, the environment can choose to (1) set up the CRS and then talk to the committer, or (2) talk to the committer and then set up the CRS. For simplicity, we streamline these interleavings, which, in turn, leaves fewer cases to reason about in the proof.

This is the reason why, for example, we added the code from Line 4 to Line 9 to the protocol of Listing 3. Of course, changes in the target behaviour propagate to the source, and thus our simulator in Listing 6 also contains code that reduces interleavings.

Z **talks to CRS:** One of the interleavings that we fixed is the environment talking to the CRS, which is an event that must happen prior to the protocol itself. The reason this must happen

has to deal with one of the cases of the proof. When the sender party is corrupted, it forwards all messages from the environment. Thus, we have two cases: either the environment behaves like an honest sender, or it does not. In the former case, we need to know that the environment has communicated with the CRS in order to derive the goodness of the values that are being sent. Essentially, if we do not force this communication, then we have more cases to consider for the environment to not behave like a honest sender. Thus, to simplify the proof, we fixed the communication to happen before the protocol.

Security argument: One simplifying assumption of our model is the symbolic crypto model that we have added into RILC (in Section 5.2.1). This common assumption [8, 9, 88, 93] ensures that the security argument we need to consider is solely about the randomness distributions in traces.

Specifically, when unravelling the reductions of the protocol, we obtain the following traces. Note that for simplicity, we present them in symbolic form, that is, instead of listing all the single-bit combinations, we represent their values abstractly, with a symbol: Also, we elide the trace of events between the environment and the Crs of Listing 4 with ....

- ((CrptNo)? · (Ok)! · ... · (Commit b)? · (Receipt)! · (Open)? · (Opened b)!, 1/2)
- ((CrptP)? · (Ok)! · ... · (Commit′ y)? · (Receipt)! · (Open′ b r)? · (Opened b)!, $1/2 \times 1/\varphi$)
- ((CrptP)? · (Ok)! · ... · (Commit′ y)? · (Receipt)! · (Open′ b r)? · (Error)!, $1/2 \times 1/(1 - \varphi)$)

As we can see, the total of these traces amount to 1, so they capture all the behaviour of the protocol. In these traces, b and y are single-bit values (i.e., either 0 or 1), while r is a λ-long sequence of bits. The probabilities are the most interesting bit: the first $1/2$ comes from the environment choice on the corruption of P, i.e., that is the probability of the environment sending the first message as a CrptNo or as a CrptP. Probability $1/\varphi$ is the probability of the environment guessing both b and r.

Following the steps mentioned above, we follow the source reductions and see that the source modules (simulator, ideal functionality, dummy parties and fake CRS) perform the same traces, at least wrt the emitted actions. We then have to argue that the distribution of all target traces matches the source one. Wrt the choice of the initial message, they are chosen by the environment using the same message distribution, so the first $1/2$ probability exists in source traces too. Wrt the probability of the environment guessing b and r, we notice that the environment has the same probability of generating those values, since source and target values come from the same set, and since the environment does not know more in the source than in the target.

Thus we can conclude that the source does exactly the same traces presented above.

6.2.5 Excess Code in Protocol. After seeing the amount of code that is required in order to apply our approach, we see the protocol code having too much boilerplate code as a possible criticism. We agree with this point, and we believe to be justified in having added said boolerplate, given the simplification they provide. Moreover, it is worth noting that the additional boilerplate surrounds the original protocol code and thus can be morally separated from it.

Some of the boilerplate code has been added as part of the attacker model (e.g., Line 2 in Listing 3 where the corruption message is received) and that could have been handled differently. For example, given the static corruption setting, we may want to say that in order to prove that a protocol UC-realises a functionality, one has to provide different implementations: one for the protocol and one for each of the corruption cases considered. That would simplify the presentation of the protocol, and protocol implementors would benefit from that.

The additional synchronisation messages provide another part of the boilerplate, but we believe that they simplify the pen-and-paper approach significantly, since there would many interleavings to consider. An alternative would be to fix a scheduling of messages, and prove the protocol UC-realises a functionality given a certain scheduling. Finally, as we move to using tools to mechanise these proofs, that boilerplate may not be necessary. As we report in Section 7, our encoding of this proof in DEEPSEC lets us strip the protocol, the CRS, and the simulator of the synchronisation messages.

6.3 UC Commiments as *RC* Proofs for the Adaptive Corruption Model

For the adaptive corruption setting, we need to adapt both the ideal functionality and the protocol. The adaptive setting is a difficult one for commitments: it is known that adaptive security requires either an exotic assumption (e.g., securely erasable memory) or weaker security properties (Hirt et al. [59, §1.2] discuss in recent work that sidesteps this issue). For simplicity we take the latter approach, demonstrating a scheme that sacrifices the hiding property. Thus, we start this section with the new ideal functionality (Section 6.3.1). Then we present the protocol (Section 6.3.2), followed by the simulator (Section 6.3.3) and the proof itself (Section 6.3.4). As before, this proof has also be mechanised in DEEPSEC, which we report in the next section.

6.3.1 Ideal Functionality. The only difference between the ideal functionality for the adaptive corruption case (Listing 10) and the one for the static corruption case (Listing 1) are the additional messages on Lines 3 and 4. The functionality immediately leaks the commitment bit (Line 3) and then waits for input to resume the commitment (Line 4).

```
1 fob =
2   rd Commit b from Read(PF)      // receive the commitment bit
3   wr Leak b to Write(FS)         // immediately leak it
4   rd DoCommit from Read(PF)      // resume the functionality as before
5   wr Receipt to Write(FS)
6   rd Open from Read(PF)
7   wr Opened b to Write(FS)
```

Listing 10. Ideal functionality for an only-binding commitment.

6.3.2 Protocol (and CRS). As before, we enrich all real-world parties (i.e., the committer, the receiver, and the CRS) with additional synchronisation messages, to limit the amount of interleavings we have to deal with in the proof. The first party that starts is the CRS, which awaits a message from the environment and then generates the CRS as before in Listing 3. Then it sends the public strings message to the environment first, then to the committer, and finally to the receiver. The CRS acts as the synchronisation party between committer and receiver, with the additional messages WaitCrs, Waited, SynchOpen, Synched, SynchOpen′, and Synched′.

The committer (P) and the receiver (Q) protocols are very similar to their static-corruption counterparts save for one key difference. There is no communication that happens on authenticated channels directly between P and Q: all protocol-related messages are transmitted on an attacker interface. In our convention, that is visible in the channel names used in the communication: they are of the form AP, PA, AQ or QA. This indicates that the message sent or received by the protocol party goes through the attacker (and thus possibly the environment) before reaching the other party.

In the protocol description we comment the additional lines that have been added for synchronisation, as well as those lines that have been added to give up the hiding property.

```
CRS =
  rd GetCrs from Read(ZCrs)
  let s = takernd
  let r0 = takernd
  let r1 = takernd
  let pk0, _ = keygen(r0)
  let pk1, _ = keygen(r1)
  wr WaitCrs to Write(CrsP)                 // start the committer
  rd Waited from Read(PCrs)                 // ack the committer started
  wr PublicStrings s pk0 pk1 to Write(CrsZ) // send values to environment
  rd Getcrs from Read(PCrs)                 // ack the committer wants the CRS
  wr WaitCrs to Write(CrsQ)                 // start the receiver
  rd Waited from Read(QCrs)                 // ack the receiver started
  wr PublicStrings s pk0 pk1 to Write(CrsP) // send values to the committer
  rd GetCrs from Read(QCrs)                 // ack the receiver wants the CRS
  wr SyncOpen to Write(CrsP)                // tell the committer it can open
                                            //  the commitment
  rd Synched from Read(PCrs)                // ack the committer can open
  wr PublicStrings s pk0 pk1 to Write(CrsQ) // send values to the receiver
  rd SynchOpen' from Read(PCrs)             // ack the committe can forward the open
                                            //  to the receiver
  wr SynchOpen to Write(CrsQ)               // tell the receiver it can open
                                            //  the commitment
  rd Synched from Read(QCrs)                // ack the receiver can open
  wr Synched' to Write(CrsP)                // tell the committer to send
                                            //    the open message
```

Listing 11. CRS for adaptive corruption.

6.3.3 Simulator. The simulator follows the same structure of the previous one, save for one detail. Since the value of the committed bit is known, the simulator does not have to rely on inverting the *prg* used by the *(fake)CRS* in order to know what the commitment bit is.

The fake CRS (Listing 14) is essentially the same as before save for the different synchronisation messages, which we indicate with a comment.

Since there are more messages being exchanged between the environment and the functionality, the dummy parties (Listing 15) contain additional proxy messages. For simplicity, some of these messages are forwarded to the simulator, as in Lines 7 and 9. Those are messages that the environment should send to the simulator directly. To avoid having the simulator listen on the environment channel, and then synchronise with the dummy party, we let the dummy parties be the only entities that talk to the environment. The dummy parties then forward to the simulator those messages that are not meant for the simulator (in this case, the *Commit′* and the *Open′* messages).

As before, the interfaces of the linked source parties (dummy P, dummy Q, S, *FAKECRS* and *fob*) are the same as the interfaces of the linked target ones (P, Q, and CRS).

6.3.4 RC Proof.

Definition 13 (Compiler for the Adaptive Corruption Commitment)**.**

$$[\![\cdot]\!]^{adap}_{comm} \stackrel{\text{def}}{=} [\![P_{comm}]\!] \rightarrowtail \mathbf{P_{comm}} \qquad \text{where } P_{comm} \text{ is Listing 10},\ \mathbf{P_{comm}} \text{ is Listing 12}$$

Theorem 11 (The $[\![\cdot]\!]^{adap}_{comm}$ Compiler is *RHP*)**.**

$$\vdash [\![\cdot]\!]^{adap}_{comm} : RHP$$

The proof follows the same structure of the one presented in Section 6.2.4, so we leave its details for the technical report [86]. The traces generated in the adaptive corruption case contain more

```
processes P =
  rd WaitCrs from Read(CrsP)          // added synchronisation
  wr Waited to Write(PCrs)            // added synchronisation
  rd Commit b from Read(ZP)
  wr Leak b to Write(PA)              // added for no hiding
  rd DoCommit from Read(AP)           // added for no hiding
  wr GetCRS to Write(PCrs)
  rd PublicStrings s pk0 pk1 from Read(CrsP)
  let r = takernd
  let y = (if b == 0 then prg pk0 r else xors(prg(pk1, r), s))
  wr Commit' y to Write(PA)
  rd SyncOpen from Read(CrsP)         // added synchronisation
  wr Synched to Write(PCrs)           // added synchronisation
  rd Open from Read(AP)
  wr SyncOpen' to Write(PCrs)         // added synchronisation
  rd Synched' from Read(CrsP)         // added synchronisation
  wr Open' b r to Write(PA)

Q =
  rd WaitCrs from Read (CrsQ)         // added synchronisation
  wr Waited to Write(QCrs)            // added synchronisation
  rd Commit' y from Read(AQ)
  wr GetCRS to Write(QCrs)
  rd PublicStrings s pk0 pk1 from Read(CrsQ)
  wr Receipt to Write(QA)
  rd SynchOpen from Read(CrsQ)        // added synchronisation
  wr Synched to Write(QCrs)           // added synchronisation
  rd Open' b r from Read(AQ)
  if (b == 0 && y == prg(pk0, r)) \/ (b == 1 && y == xors(prg(pk1, r), s))
    wr Opened b to Write(QZ)
  else
    error
```

Listing 12. Only-binding commitment protocol for adaptive corruption.

```
S =
  rd StartS from Read(CrsS)
  wr FakeSetup from Write(SCrs)
  rd Fake pk0 pk1 td0 td1 s from Read(CrsS)
  wr Ok to Write(SCrs)
  rd Leak b from Read(FS)
  wr Leak b to Write(SQ)
  rd Receipt from Read(FS)
  let r = takernd
  let y = if b == 0 then prg pk0 r else xors(prg(pk1, r), s)
  wr Commit' y to Write(SQ)
  rd Commit' y' from Read(PS)
  wr Receipt to Write(SQ)
  rd Opened b from Read(FS)
  wr Open' b r to Write(SQ)
  rd Open' b' r' from Read(PS)
  if (y ==y') && (b == b') && (r == r')
    wr Opened b to Write(SQ)
  else
    error
```

Listing 13. Simulator for adaptive corruption.

actions than the previous ones, as presented below. For simplicity, we only show the action trace, without the probabilities, and abstract the values symbolically. We also elide the error trace where

```
FAKECRS =
  rd GetCrs from Read(ZCrs)          // added synchronisation
  wr StartS to Write(CrsS)           // added synchronisation
  rd !FakeSetup from Read(SCrs)
  let pk0, td0 = keygen([1..1])
  let pk1, td1 = keygen([1..1])
  let r0 = takernd
  let r1 = takernd
  let s = xors(prg(pk0, r0), prg(pk1, r1))
  wr Fake pk0 pk1 td0 td1 s to Write(CrsS)
  rd Ok from Read(SCrs)
  wr WaitCrs to Write(CrsP)        // added synchronisation
  rd Waited from Read(PCrs)        // added synchronisation
  wr PublicStrings s pk0 pk1 to Write(CrsZ)
```

Listing 14. Fake CRS for adaptive corruption.

```
P =
  rd WaitCrs from Read(CrsP)
  wr Waited to Write(PCrs)
  loop(2)(
    fwd(Read(ZP))(Write(PF))  // forward from environment to functionality
  )
  fwd(Read(ZP))(Write(PS))    // forward from environment to simulator
  fwd(Read(ZP))(Write(PF))    // forward from environment to functionality
  fwd(Read(ZP))(Write(PS))    // forward from environment to simulator

Q =
  loop(5)(
    fwd(Read(SQ))(Write(QZ))
  )
```

Listing 15. Simulator Parties for adaptive corruption.

the last action is replaced by an error message. This case happens when the environment does not act as a forwarder for the *Commit′* and the *Open′* messages, but in the case it acts maliciously and tampers with the communicated y, b and r values.

- $\mathit{GetCrs}?\cdot\mathit{PublicStrings\ s\ pk0\ pk1}!\cdot\mathit{Commit\ b}?\cdot\mathit{Receipt\ b}!\cdot\mathit{DoCommit}?\cdot\mathit{Commit'\ y}!\cdot\mathit{Commit'\ y'}?\cdot \mathit{Receipt}!\cdot\mathit{Open}?\cdot\mathit{Open'\ b\ r}!\cdot\mathit{Open'\ b'\ r'}?\cdot\mathit{Opened\ b}!$

Applying the backtranslation proof technique to this trace yields the following elements of the simulator *S*, of the *FAKECRS*, and of the proxies *P* and *Q*:

- **GetCrs?** gets backtranslated to Line 2 of *FAKECRS*
- **PublicStrings s pk0 pk1!** is Line 14 of *FAKECRS*
- **Commit b?** is Line 5 in *P* (iteration 1 of the loop)
- **Receipt b!** is Line 13 in *Q* (iteration 1 of the loop)
- **DoCommit?** is Line 5 in *P* (iteration 2 of the loop)
- **Commit′ y!** is Line 13 in *Q* (iteration 2 of the loop)
- **Commit′ y′?** is Line 7 in *P*
- **Receipt!** is Line 13 in *Q* (iteration 3 of the loop)
- **Open?** is Line 8 in *P*
- **Open′ b r!** is Line 13 in *Q* (iteration 4 of the loop)
- **Open′ b′ r′?** is Line 9 in *P*
- **Opened b!** is Line 13 in *Q* (iteration 5 of the loop)

This initial part of the backtranslation essentially builds all of the proxies, as well as the entry and the exit point of the *FAKECRS*.

At this point, since we know that there must only be one party computing at each time, we can follow the trail of messages in order to start filling out the rest of the missing code. For example, fixing the interleavings between the parties and the *FAKECRS* creates Lines 2, 3, 4 in the *FAKECRS* and the related Lines 2, 3 in *S*. The bulk of the *FAKECRS* (Lines 5 to 9) cannot be automatically generated by the backtranslation, but Lines 10 to 13 follow directly from having fixed the interleavings. Thus, once Lines 10 to 13 in *FAKECRS* are crafted, the related lines (Line 4, 5 in *S*, and Lines 2, 3 in *P*) are also a straightforward addition. As another example of straightforward code generation, notice that message **Commit b?** is forwarded from the proxy of *P* to the functionality, and then it causes a *Leak* message to be sent into the simulator. This generates Lines 6 and 7 in *S*, and one iteration of the loop in *Q*.

With this procedure, this specialised trace-based backtranslation proof technique has built all of the code in *S*, *P*, *Q* and *FAKECRS* except for Lines 9 and 10 in *S* and Lines 5 to 9 in *FAKECRS*. This is expected, and in secure compilation work that use this proof technique, a bit of creativity is also needed in order to make the proof go through.[10] We envision that, with the proof automation described in Section 7, one can employ program synthesis techniques to automate the search space for the missing creative bits, but we leave investigating this for future work.

One interesting insight we encountered while doing said proof was the inability to complete it while trying to retain both the binding and hiding properties. In fact, there comes a time (Line 11) when performing the reductions that the simulator must create the commitment value (y in the code). If the protocol is trying to retain the hiding property, the simulator has no data to conjure y from, so the proof gets stuck. On the other hand, if the protocol gives up hiding, it knows enough data to create such y, as done in Line 10.

The proofs provided in this section already showcase how to make good use of our connection in order to obtain rigorous, formal UC proofs. However, as we point out, many of the steps of these proofs can be automated, and this insight is what we expand upon in the next section, where we mechanise the proofs presented here.

7 MECHANISING THESE PROOFS

This section reports on how to mechanise the proofs of the previous section. First, this section justifies the choice of the tool for this mechanisation: DEEPSEC (Section 7.1). Then it details how to model protocols in DEEPSEC, i.e., how to go from the pen-and-paper protocol definitions of the previous section to the internal representation of DEEPSEC (Section 7.2). Finally, this section reports the lessons learned while doing this mechanisation effort (Section 7.3).

7.1 Picking the Tool

To explore the potential for mechanisation of the previous proofs, we investigated the applicability of off-the-shelf tools for the analysis of the previous case study. Our requirements were twofold. First, the tool must provide a high-degree of automation, since the proofs required to manually check the semantics of each program reduction. Second, the tool must rely on a language whose communication model is similar to the one of RILC. We could not use existing ILC-based tools since they are currently limited to type-checking processes, with no symbolic execution of processes. Since we are interested in knowing what is *possible* from a mechanisation perspective, we accept a (small) semantic gap between RILC and the native language of the tool.

[10] For example, code that operates type conversion needs to be crafted and added by the backtranslation for the proof of RSC of a compiler between typed and untyped code [83].

Thus, our tool choice fell on DEEPSEC [44], which is a decision procedure for process equivalence in the applied-pi calculus from the perspective of a Dolev-Yao attacker (or, environment, in RILC terms). The Dolev-Yao attacker is an unusual choice in the context of UC, but our focus is on the possibility of fully automating the trace-based analysis. Künnemann, Patrignani, and Cecchetti [68] perform a similar analysis in the computational model and employ CryptoVerif as a backend. Even though CryptoVerif is known for its comparatively high degree of automation, their case study requires manual guidance. Of course, CryptoVerif has a probabilistic semantics, which is harder to reason about than DEEPSEC's non-deterministic, non-probabilistic execution model. In this work, however, we argue that automation is possible, once suitable abstractions have been established.

DEEPSEC represents protocols in the applied-π calculus, whose semantics is fairly close to the one of RILC and thus omitted. The semantics of applied π has three notable limitations. First, applied π does not have the cryptographic primitives required by the protocol (i.e., no *xors*, no *prg*, etc.). Second, applied π lacks an explicit module system like the one of RILC. Finally, applied π processes are not necessarily confluent. We need to account for these while modelling protocols and we explain in detail how we do this accounting in the next section.

Another limitation is that the decision procedure of DEEPSEC is undecidable [3], so the tool needs to bound the number of processes running in parallel by some fixed number. Fortunately, the commit protocol we chose as fixed in Section 6.2 only covers a single session, so our verification is not affected by this limitation.

7.2 Modelling Protocols in DEEPSEC

We now describe the choices behind modelling channels, primitives, and the various entities of the protocol.

Modelling Channels. DEEPSEC channels are one-directional, untyped channels, akin to RILC channel ends, which can be labelled as public or private. Communication over public channels ends up in DEEPSEC traces while communication over private ones does not. Thus, we set up one DEEPSEC channel for each channel end in Section 6.2, that is, one channel between all pairs of protocol, subprotocol, environment and attacker that talk to each other. The channels that interact with the environment are public, while those that do not are private.

In general, any channel whose read and write ends are both specified in the protocol should be private, to ensure communication on that channel does not generate any trace. Dually, if at least one of the read and write ends is not specified in the protocol, that channel should be public, since it models communication with the environment.

Modelling Primitives. The first limitation of DEEPSEC is the lack of built-in cryptographic primitives that the commitment protocol relies on. This is a design choice of DEEPSEC, and the tool lets programmers define arbitrary primitives (via keyword `fun`) and their reductions (via keyword `reduc`). Thus we add reductions for the *prg* and its inversion, as well as for *xors* (i.e., for those rules added in Section 5.2.1).

```
1 fun prg/2.
2 fun keygen/1.
3 reduc invert(prg(keygen(trapdoor),seed),trapdoor) → seed.
4
5 fun xor/2.
6 reduc dexor1(a,xor(a,b)) → b.
7 reduc dexor2(b,xor(a,b)) → a.
```

Observe that the XOR theory is incomplete. It is missing the associativity, commutativity, the neutrality of 1 and elimination property of 0. As Deepsec cannot handle equational theories, we had to simplify XOR in this regard.

Modelling Entities as Processes. The second limitation of applied π is its lack of a module system. To overcome this, we identify applied π processes to have the same role as RILC modules. Thus we use process composition (|) as module composition ($\bowtie$), since they behave in the same way.

We code a process for each of the entity in Section 6.2, in the real world we have `committer` (for P), `receiver` (for Q), and `CRS`, while in the ideal world we have the ideal functionality `COM` (for F), and then `dummy-committer` (for P), `dummy-receiver` (for Q), `fakeCRS` (for *FAKECRS*), and `simulator` (for S). For the sake of space, we do not show the code of these processes here, we provide a list of all processes at the end of this section, together with a link to the code.

Below we present the code `COM`, which we use to highlight how DEEPSEC processes are essentially a complete transliteration (modulo syntactic difference) of their RILC counterparts.

```
let COM =
    in(p2f,x22); let (=Commit,b) = x22 in
    out(f2a,Receipt);
    in(p2f,x23); let =Open = x23 in
    out(f2a,(Opened,b)).
```

The statement `in(p2f,x22)` reads from a channel from the dummy party (`p`) to the functionality (`f`) and binds the result into variable `x22`. Then `let (=Commit,b) = x22 in` checks that the message in `x22` has the form `(=Commit, b)`, for some value `b`. This is just like `rd Commit b from Read(PF)` in Listing 1. Dually, `out(f2a,Receipt)` writes to the channel from functionality (`f`) to simulator (`a`), just like `wr Receipt to Write(FS)` in RILC (cfr Listing 1).

Unfortunately, the previous list of processes is not sufficient to successfully model the commitment protocol in DEEPSEC because of the last limitation of the tool: its lack of confluence. As they stand, there are interleavings of the execution that complicate the reasoning of trace equivalence.

Thus, we introduce another process, essentially an environment wrapper between the environment in DEEPSEC and the processes we have defined. This environment wrapper fixes the flow of messages from the environment, essentially forcing the scheduler policy, making the wrapped processes confluent. This, in turn, eliminates the need to synchronise different entities (as done in Sections 6.2 and 6.3), since the environment wrapper provides a synchronisation entity already. The environment wrapper has the following code, which is an input-output alternation (we use the variable names to indicate what message is expected to be passed each time):

```
let env =
    in(z2e,x30corrupt);
    out(e2p,x30corrupt);
    in(p2e,x31ok);
    out(e2z,x31ok);
    in(z2e,x32getcrs);
    out(e2s,x32getcrs);
    in(s2e,x33pubstrings);
    out(e2z,x33pubstrings);
    in(z2e,x34commit);
    out(e2p,x34commit);
    in(q2e,x35receipt);
    out(e2z,x35receipt);
    in(z2e,x36open);
    out(e2p,x36open);
    in(q2e,x37opened);
    out(e2z,x37opened);
    0.
```

All wrapped processes talk to the environment via channel ends that end with `e`, for example an input channel from the environment to the simulator is called `e2s`. The environment wrapper provides a binding for all those channels and forwards them through `e2z` or `z2e` channels, where `z` is used to identify the DEEPSEC environment. The channels between the environment wrapper and all wrapped processes are set to private.

We compose the real and ideal world from these processes as follows:

```
let realw = env | committer | receiver | CRS.
let idealw = env |  dummy_committer | dummy_receiver | COM | sim | fakecrs.
```

In order to test that `realw` and `idealw` are trace equivalent, we issue the following DEEPSEC command, which returns true, attesting to the equivalence of the two systems.

```
query trace_equiv(idealw,realw).
```

The DEEPSEC files can be found at the project page, https://uc-is-sc.github.io/, specifically at:

https://uc-is-sc.github.io/deepsec/deepsec-commitment.zip

and they include the following files. Note that for simplicity, we split the proof of the corruption case in two files, one where the no-corruption case, and one for the case of the corruption of the committer:

`com-nocor.dps`: contains the proof for the static corruption case when no party is corrupted;
`com-corP.dps`: contains the proof for the static corruption case when the committer is corrupted;
`com-adaptive.dps`: contains the proof for the adaptive corruption case.

7.3 Lessons Learned

One of the benefits of adding the `env` process is that we can debug the trace equivalence proof step-by-step. Essentially, the `env` process is a sequence of input-output pairs that identies what the environment is sending and what it is reading from whatever world (be it real or ideal) it is communicating with. By considering such input–output pairs incrementally (i.e., first the first pair, then the first two pairs and so forth), we can incrementally check that trace equivalence holds. And when it does not hold, we know precisely at which step did the equivalence fail. We found "attacks" at almost every step, about half of them due simple typos, and the rest of them pointing out issues with the setup or the simulator related to the scheduling messages meant to achieve confluence.

The graphical user interface of DEEPSEC allowed us to step through counter-examples in both real and ideal world, identifying the source of these issues. Analysis took a few seconds, so we could often fix them by trial-and-error.

In summary, the symbolic analysis of DEEPSEC is an excellent starting point for mechanisation, and the automation obtained due to the Dolev-Yao model is very useful. Even though a cryptographic analysis would give stronger result, we observe that the majority of messages the environment receives is literally the same. Hence a hybrid approach that combines cryptographic reduction with symbolic execution could be beneficial.

As previously mentioned, the confluence requirement of RILC results in unnecessarily complicated and yields unnatural models of protocols. This is a drawback that RILC inherits from the classical UC frameworks. While confluence simplifies the analysis (there are fewer traces to match between the real and ideal world, even though they are longer), DEEPSEC was designed specifically to handle asynchronous communication, heavily exploiting parallel computing. Comparing our model with standard DEEPSEC models, we see no reason why DEEPSEC could not handle similar models with real asynchrony. We are thus confident that future verification tools could easily draw from the algorithm of DEEPSEC to handle asynchronous, non-deterministic communication.

Compared to RILC, this would improve the generality and readability of the protocol models we analyse.

8 DISCUSSION

This section discusses the relation of our results to other notions in UC and *RC*. First, we briefly discuss other UC notions and how they respect the axioms of Section 3 (Section 8.1). Then we discuss why our connection is the most precise, by showing that no other *RC* notion connects with UC, and that an existing connecting conjecture is false (Section 8.2). Finally, we argue the different choice of programming language style (reactive VS non-reactive, Section 8.3).

8.1 Different UC Notions

Several closely related notions of composable, simulation-based security exist in the literature, including GNUC [60], IITM [67], EasyUC [43], ILC [73], iUC [36], among others. In this section we argue that the four axioms of Section 3, which that section discusses in the context of UC as defined by Canetti [37], also hold in all of the previously mentioned models (and likely hold in other models from the literature). As in the rest of this paper, we leave the many issues raised by *computational* notions of UC to future work.

To begin, we note that Axiom 1 and Axiom 3 hold by inspection for all of these systems: these two axioms merely require defining a trace model, ExecT, and Exec commensurate with the computational model of the underlying notion of UC.

Axiom 4—the fact that non-probabilistic environments are complete—holds in all of these systems for the same reason it does in the UC notion of Canetti [37]: in all cases, the definition of simulation universally quantifies the environment *after* binding the protocol and the adversary. If there exists a probabilistic distinguishing environment, then there exists at least one set of random choices for which the environment succeeds, which we can hard-code into a deterministic environment. For GNUC, see [60, Def. 7]; for IITM, see [67, Def. 1]; for ILC, see [73, §6.3]; for iUC, see [36, p. 8]. EasyUC mirrors Canetti's UC definition in this regard, and Canetti explicitly states that non-probabilistic environments are complete [37, p. 47].

Axiom 2—the fact that finite traces contain the final bit—also holds in all of these systems. In GNUC, the environment's output is an arbitrary string, which is post-processed by a distinguisher that returns a bit [60, Defs. 6 and 7]; this distinguisher is exactly the extraction function β of Axiom 4. In IITM, either the environment writes a single bit to a `decision` tape or execution halts without writing such a bit, in which case the result is 0 by definition; the same holds in iUC [36, p. 7]. In ILC, execUC outputs a single bit [73, §6.3]. In both EasyUC and Canetti's UC, the environment terminates the trace with a final bit.

More generally, the axioms of Section 3 appear to hold because of the *structure* of composable security—the execution model, quantifier ordering, and distinguishing procedure. Thus, we expect these axioms to hold for essentially all related notions of composable security.

8.2 Wrong Connections

RHP is the only criterion we could have chosen to instantiate our connection. Other candidates include other *RC* criteria from the work of Abate et al. [14], as well as a conjecture from Hicks [58]: fully-abstract compilation (*FAC*) [1]. This section first debunks the connection with other *RC* criteria (Section 8.2.1) before disproving the *FAC* conjecture (Section 8.2.2).

8.2.1 RHP and Other RC Criteria. Abate et al. [14] provide *RC* criteria for preserving all known classes of hyperproperties, and arrange them in a partial order on their expressiveness. In order to

show that *RHP*, and only *RHP*, coincides with UC, we prove that all neighbouring elements to *RHP* do not coincide with UC. These elements are:

- *RSCHP*, the preservation of subset-closed hyperproperties (*SCHP*);
- *RTP*, the preservation of arbitrary trace properties, which has been proven to be strictly weaker than *RHP*.

RSCHP Also Works. SCHP intuitively formalises the notion of refinement, and it has been proven to be what correct compilers preserve through compilation (in absence of attackers though) [12].

Arbitrary hyperproperties are defined as sets of sets of traces, a hyperproperty H is *SCHP* if for any set of set of traces b in H, then H also contains all subsets of b. Formally:

$$SCHP \stackrel{\text{def}}{=} \{H \mid \forall b' \subseteq b.\ \text{if } b \in H \text{ then } b' \in H\}$$

There is a single difference between *RHP* and *RSCHP* (below): while the former is a refinement notion, the latter is not. Instead, it is a coincidence, as the target and source behaviours are connected by a co-implication.

Definition 14 (Robust Subset-Closed Hyperproperty Preservation)**.**

$$[\![\cdot]\!] \vdash RSCHP \stackrel{\text{def}}{=} \forall P, \mathbf{A}.\ \exists A.\ \mathbf{Behav}(\mathbf{A} \bowtie [\![P]\!]) \subseteq Behav(A \bowtie P)$$
$$\text{or equivalently} : \forall P, \mathbf{A}.\ \exists A.\ \forall \bar{t}.\ \text{ if } \mathbf{A} \bowtie [\![P]\!] \rightsquigarrow \bar{t} \text{ then } A \bowtie P \rightsquigarrow \bar{t}$$

Interestingly, this notion also coincides with UC (proven in Isabelle/HOL in Theorem `RSCHPtoUC` and Theorem `UCtoRSCHP`). This suggests that with the UC trace model, arbitrary hyperproperties and subset-closed ones collapse. In fact, traces in UC contain probabilities, and given a set of traces, the sum of their probabilities must be exactly 1. *RSCHP* mandates that the traces of one system ($\mathbf{A} \bowtie [\![P]\!]$) are contained in the other ($A \bowtie P$), so the latter cannot contain any additional trace with non-zero probability. Given that there cannot be any trace with zero probability (they cannot be emitted by any program, ever), the two sets must coincide, and thus the implication in *RSCHP* is equivalent to the co-implication of *RHP*.

Similar collapses of classes of hyperproperties are not novel, and there can be language operators that cause it (as discussed in Abate et al. [14]). Since they deter from the main point of the paper, we leave investigating additional collapses and the ramifications of this collapse for future work.

Other notions: Preserving Trace Properties, Hypersafety Properties and Relaxed Hypersafety Properties. Trace properties, formally defined as sets of traces [2, 71, 92], capture many functional properties of programs. Intuitively a compiler attains *RTP* if it preserves arbitrary trace properties through compilation [14].

The difference between *RHP* and *RTP* (below) is in the order of quantifiers: in the former the traces are quantified last, while in the latter, they are quantified *before* the source context.

Definition 15 (Robust Trace-Preserving Compilation)**.**

$$[\![\cdot]\!] \vdash RTP \stackrel{\text{def}}{=} \forall P, \mathbf{A}, \bar{t}.\ \exists A.\ \text{ if } \mathbf{A} \bowtie [\![P]\!] \rightsquigarrow \bar{t} \text{ then } A \bowtie P \rightsquigarrow \bar{t}$$

However, as reported by Datta et al. [46], there exists a UC-like notion whose quantifiers are in the same order as *RTP*: Reactive Simulatability (RS) [23]. Indeed, at the formal level, the only thing that changes between RS and UC is the quantifiers ordering:

Definition 16 (Reactive Simulatability)**.**

$$\pi \vdash_{\mathrm{RS}} \mathsf{F} \stackrel{\text{def}}{=} \forall \mathsf{A}, \mathsf{Z}.\ \exists \mathsf{S}.\ \textsc{Exec}[\mathsf{Z}, \mathsf{A}, \pi] \approx \textsc{Exec}[\mathsf{Z}, \mathsf{S}, \mathsf{F}]$$

Kuesters et al. [67] state that RS and UC coincide for IITM but not for the notion of UC of Canetti [37]. This would imply that *RTP* and *RHP* also coincide, given additional setting-specific assumptions. We believe the connection between RS and *RTP* can be formally proven, though we leave investigating such specific assumptions for future work.

Additional collapses whose investigation we also leave for future work include arbitrary hypersafety properties and relaxed hypersafety properties. Arbitrary *HSP* are hypersafety properties, which include meaningful security properties such as many flavours of non-interference [45]. While safety properties are identified by a set of bad prefixes (that the safety property does not extend), hypersafety properties are identified by finite sets of sets of bad prefixes (which the hypersafety property does not extend). Canonically, *HSP* are defined as finite sets of sets of prefixes, and in the finiteness of the first set lies the difference with *SCHP*. Assume we relax the notion of hypersafety to encompass *infinite* sets of finite prefixes and call this set relaXed Hypersafety (*XHSP*)

We conjecture that in the specific trace model of UC, *HSP* and *XHSP* coincide, since communication is bounded, value size is bounded and thus there cannot be an infinite set of traces. We believe that *XHSP* and *SCHP* collapse with the additional common assumptions on the trace model used in UC. This would just serve the purpose of illustrating what class does UC morally correspond to, so we leave this for future work.

8.2.2 Disproving The FAC Conjecture. Fully-abstract compilation (*FAC*) has been the de-facto standard for secure compilation until the recent proposal of *RC* [80]. Unlike *RC*, *FAC* does not rely on a notion of traces, but on the notion of contextual equivalence [89] in order to specify security properties. Two programs P_1 and P_2 are contextually equivalent (indicated as $P_1 \simeq_{\text{ctx}} P_2$) if they co-terminate no matter what attackers (program contexts) they link against. Formally, if we indicate single reduction steps as $\hookrightarrow$ and n such reductions as $\hookrightarrow^n$, termination of a (whole) program is defined as follows:

$$W\Downarrow \stackrel{\text{def}}{=} \exists n.\ W \hookrightarrow^1 \cdots \hookrightarrow^n W' \not\hookrightarrow$$

which leads to the following definition of contextual equivalence:

$$P_1 \simeq_{\text{ctx}} P_2 \stackrel{\text{def}}{=} \forall A.\ A\bowtie P_1\Downarrow \text{ iff } A\bowtie P_2\Downarrow$$

A compiler is fully abstract (indicated as $\vdash [\![\cdot]\!] : FAC$) if it preserves (and reflects) contextual equivalence of source programs in their compiled counterparts. Formally

$$\vdash [\![\cdot]\!] : FAC \stackrel{\text{def}}{=} \forall P_1, P_2.\ P_1 \simeq_{ctx} P_2 \text{ iff } [\![P_1]\!] \simeq_{\text{ctx}} [\![P_2]\!]$$

If we unfold the definitions of source and target contextual equivalence, we obtain the following statement:

$$\forall A.\ (A\bowtie P_1\Downarrow \text{ iff } A\bowtie P_2\Downarrow) \text{ iff } \forall \mathbf{A}.\ (\mathbf{A}\bowtie [\![P_1]\!]\Downarrow \text{ iff } \mathbf{A}\bowtie [\![P_2]\!]\Downarrow)$$

Thus, like *RC*, *FAC* considers a robust notion of security, since it universally quantifies over all target program contexts.

The simplest argument against the conjecture that *FAC* is UC is counting arguments. While UC is *propositional*, in that it talks about a single entity (i.e., the protocol or the ideal functionality) in each domain (concrete and abstract), *FAC* is *relational*, in that it talks about pairs of source and target programs. As such, when trying to derive an equivalence between the two notions, it is not possible to connect all of the elements: if P_1 is F and $[\![P_1]\!]$ is π, what do P_2 and $[\![P_2]\!]$ correspond to?

More generally, *FAC* talks about *arbitrary* program equivalences, since P_1 and P_2 can be chosen arbitrarily. Instead, UC talks about preserving the expected behaviour of an ideal functionality encoded in a protocol which interacts with real-world attackers.

What about Contextual Equivalence? If we model the 'expected behaviour' as simply terminating or diverging, then we can see a similarity between the notion of contextual equivalence and UC (despite its lack of an existentially-quantified simulator). There, it is clearer what the two programs correspond to: P_1 is the protocol while P_2 is the ideal functionalty linked with the simulator.

Using contextual equivalence for proving UC-like notions has been recently investigated by Morrisett et al. [77]. We believe this approach is valid, and it can be seen as a specilisation of our result that uses termination as the environment final bit instead of traces. In fact they rely on the dummy attacker theorem whose application we justify in Section 4.2. Additionally, since contextual equivalence only talks about a single language, it can only be applied to the setting where the language of the protocol and of the ideal functionality coincide. While we have done the same here, our result is more general, and using *RHP* lets one use different languages for the protocol and for the ideal functionality.

8.3 Reactive VS Non-Reactive Languages

We have seen how the connection can be set up with a reactive language (RILC), but as we mentioned before, we believe the connection holds also for non-reactive languages.

Recall that the main difference between reactive and non-reactive languages is that reactive ones have no notion of whole program, while non-reactive ones do. A whole program has a well-defined entry point for the `main` and it cannot be effectively extended further. Since all the dependencies are resolved, and the `main` is defined, any code that is added to a whole program is essentially dead code.

In reactive languages (such as RILC), there is an explicit entity in the semantics that models the environment (a configuration with the write token: $\langle \varpi; K; \Xi; C \rangle$), and any communication with it yields a trace action. With non-reactive programs, there is no such environment entity in the semantics, so one should ask what is a trace action in this setting.

To answer this question, notice that trace actions are what gets communicated on the environment interface, so we must identify the environment in the non-reactive setting. We believe that in this setting the environment effectively gets merged with the attacker. After all, as for the dummy attacker theorem, there is no need to have two universally-quantified entities (environment and attacker), and since there is no environment in this setting, both roles are covered by the attacker. Thus, any communication on the interface between the attacker and the program needs to generate a trace action.

Notice, however, that the definition of traces is central to our development. By changing the notion of traces and how they are defined, some of the presented axioms need changing. We do believe, however, that none of the existing results break, and we now briefly discuss why.

*Theorem 3 (*UC *and RHP Coincide).* This result is untouched by the language setting.

Theorem 6 (Composition in RC). In order to derive this result we rely on a number of composition operators. First, $\bowtie$ needs not return a whole program, lest its output be not composable with any other program. Then $\circledcirc_{\mathbf{T}}^{S}$ must also not return a whole program, just a complete one, the axioms that use it need not change (Axiom 6 (FFI Recombination) and Axiom 7 (Constant Addition)). Finally, some technical machinery may be needed in order for Axiom 7 (Constant Addition) to make sense. Recall that its statement is: $P \simeq \mathbf{P}$ then $P \circledcirc_{S}^{O} P \simeq P \circledcirc_{\mathbf{T}}^{O} \mathbf{P}$. Here, the programs of the premise would be whole programs, i.e., they define a `main` and they are complete. It is therefore unclear what happens when something else links against them, so long as the new program defines the main, it can be called, otherwise it is dead code. We expect the notion of the main entry point needs to be tweaked in order for this composition to make sense and leave studying this for future work.

Theorem 8 (Dummy Attacker in RC). Without a notion of environment, the dummy attacker theorem makes little sense: we cannot replace the only attacker entity with a dummy! However, we can borrow existing results from work that used a trace model similar to those required here in order to obtain similar results to the dummy attacker theorem.

Recall that in this case we need traces to be those actions that are generated across the attacker-program interface. Such trace semantics have been studied aplenty in the context of fully-abstract trace semantics [10, 65, 81], i.e., a trace semantics that is as precise as contextual equivalence. A key element of those semantics is that they simplify reasoning by eliding the notion of attackers, which gets abstracted away. Now suppose T is equipped with a trace semantics that yields just those traces. If the trace semantics of T is fully-abstract with respect to behavioural equivalence, we can use the trace semantics and eliminate the target attacker from the theorem statement. This is essentially the same result we get in reactive languages when using the dummy attacker theorem.

With one such trace semantics for T, we conjecture that we can follow a proof technique akin to the one used in reactive languages. Essentially, from the target traces we can build the source-level attacker (or, simulator) A. Then, we will have all source elements (program and simulator) and we can calculate whether they are trace-equivalent with the target compiled program. This would be a simplification of an existing proof technique called trace-based backtranslation, which is often used in secure compilation work [11, 79, 83, 85]. We leave investigating this novel proof technique for future work.

9 RELATED WORK

Programming Languages Meets Cryptography. Researchers have tried to connect programming languages and cryptography at length, as attested by the Dagstuhl seminar of 2014 [58]. Existing results span language models as well as tools that bring the two worlds closer together.

From the language side, a number of languages provide more rigorous formalisation of those languages used by cryptographers. ILC is one such language, which the authors used as a way to encode the semantics of Interactive Turing Machines [73]. As shown, ILC is a good candidate to showcase our connection.

IPDL is another such language, which is equipped with an equational logic that can be used to reason about contextual equivalence of IPDL programs [77]. IPDL is also a good candidate for our connection, and its equational logic can likely be extended to reason about trace equivalence of IPDL programs (once they are lifted to the reactive setting). In the case where the source and target languages differ, however, the IPDL logic would not be useful, since it is bound to a single language.

To show the high-level security of some low-level cryptographic primitives researchers have come up with high-level calculi [3, 5, 7, 8, 52] or cryptographic-aware compilers [17] – a large body of work recently surveyed by Hastings et al. [57] . Interestingly, Acay et al. [17] also discovered one half of our connection (*RC* implies UC), but were not interested in the other half[11], as we showcase in this work.

When it comes to tools, there are a number of verification tools for expressing cryptographic protocols and checking some of their properties. These tools include DEEPSEC [44] EasyCrypt [28] CryptoVerif [32] and Squirrel [25]. All of them have a reactive formal language as the semantic foundation for their protocol specification language, but none is really built to prove UC. With the results of this work, we demonstrate, in general, that tools that are built to prove (trace) equivalences can be used to provide UC mechanised proofs.

UC *Works.* Universal composition was introduced in the seminal work of Canetti [37], but the idea of describing security via an ideal functionality that describes all non-attacking network

[11]Personal communication.

traces via simulation goes back much further [54]. We already discussed UC and some of its descendants [36, 73]. Our work is a generalisation of this concept that removes the need to specify the communication down to the machine model. Our main result relates robust compilation to perfect emulation. For a detailed discussion of the challenges concerning *computational* emulation, we refer to Hofheinz et al. [63].

Concepts from universal composability were transferred from ITMs to other languages, but also embeddings between different UC-like frameworks have been shown (e.g., UC [37] into IITM [90]). One line of research considers the intricacies of UC in the applied-pi calculus [33, 48]. This line of research formulates emulation using process equivalences in the applied-pi calculus and composition using a rewriting of the channels used inside a process. This requires to track which channels are used for network communication, which for the environment and which are external. This requires several well-formedness conditions on the processes and complicates the definitions, while essentially describing the interfaces at the meta level rather than inside of the language. By contrast, the composition operator for RILC (Section 5.2.2) benefits from a module system, combining defined interfaces in a straight-forward way. In the end, both types of composition behave very similarly. The former approach is directly compatible with existing tools such as ProVerif and DEEPSEC (which do not support a module system), albeit the well-formedness conditions need to be checked manually. Our approach is conceptually simpler and separates more clearly between language semantics, module system and meta theory. In terms of programming languages, UC has been adopted to a fragment of Java [70] and (concepts like ideal functionality and emulation) to a fragment of F# [53]. Each of these works considers UC (or related concepts) within their respective target language. Viewing UC as a form of robust compilation stands to simplify deriving similar results for other languages and help relate these results across languages.

The results just mentioned come with entirely different verification methods, the former using program dependency graphs to obtain emulation from non-interference [70], the latter using refinement types to obtain assert-rely-style compositionality by type-checking [53]. At the protocol level, the UC variants related to the applied-pi calculus [33, 48] rely on off-the-shelf checkers for protocol equivalences. Delaune and Hirschi [47] provided a survey about these in 2007; a more recent survey with a more general scope also contains a section on equivalence properties [26]. By and large, protocol equivalence checkers can handle prototols with an unbounded number of sessions if the 'two worlds' are structurally similar or if syntactic conditions can ensure that results for a small number of sessions translate to the unbounded model — sometimes called *small-model results*. For a bounded number of sessions, and in the Dolev-Yao model there are multiple decision procedures, providing convenience and automation, with DEEPSEC being the most recent and most advanced. This points to an interesting research question: can we structure the composition operator so that small-model results apply and we can use these decision procedures (in the Dolev-Yao model?).

In terms of direct mechanisations of UC, we already discussed two formalisations in the cryptography-centric theorem-prover EasyCrypt [27, 43] in Section 6.1.1. Constructive Cryptography, a UC-like framework, was formalised in Isabelle/HOL [30], These two are the most advanced mechanisation efforts for UC-like models in the computational model. Both require an enormous amount of effort and expertise for modelling and proving, but provide a very high degree of assurance. There is hope though: there are verification tools in the computational model that provide more automation [32] by using program transformations that are computationally justified. A related line of research justifies deduction systems that perform symbolic reasoning in the computational model [25, 29], likewise promising to increase the amount of automation. Some of these [25, 32] model protocols in a computational variant of the applied-pi calculus, suggesting that the choice between the Dolev-Yao model and the computational model is not such a fundamental questions. The translation and the

composition operators might not be all that different and the most automated reasoning tools in the computational model take inspiration from those in the Dolev-Yao model.

Datta et al. [46] present a number of UC-like notions and compare their expressiveness. This looks like a good starting point to further connect programming language notions with cryptographic ones, as hinted in Section 8.2.1.

The idea of providing high-level abstractions for cryptographic protocols in programming languages predates UC, of course. Abadi et al. [8], for example, translate a high-level language with an abstraction for channels to a low-level language that implements this channel via encrypted and channel communication. On the first glance, this provides *less* flexibility: the abstraction result corresponds not to UC as a concept, but to a singular emulation result. On the other hand, this approach allows, in principle, a tighter integration into the language. Consider, for example, communication protocols, which are better abstracted as a channel, versus cryptographic primitives like encryption, which are better abstracted to symbolic terms. Exploring whether UC results can be related to *RC* this way – which is competing to the approach presented in this work – remains an open problem.

RC Works. The theory of robust compilation was recently devised by Abate et al. [14] and later expanded to account for differences between the source and target trace models [13]. We believe our connection does not need to account for source and target trace difference since UC deals with the same language for protocols and functionalities.

A novel criterion for secure compilation, *RC* has been compared to existing secure compilation notions such as fully-abstract compilation [15]. We reported that fully abstract compilation is not a good candidate for our connection in Section 8.2.2.

RC has been used to reason about the preservation of arbitrary safety properties through compilation [83, 84] and through translation validation [35] even with mechanised proofs [51]. Additionally, it has been used to reason about the absence of leaks due to speculation attacks such as Spectre [85].

10 CONCLUSION

This paper presented a striking connection between the cryptographic framework of Universal Composability (UC) and the secure compilation criterion called Robust Hyperproperty Preservation (*RHP*). This paper first formalised (and mechanised in Isabelle) this connection and then identified a the requirements for lifting this connection to an *RHP*-compiler between arbitrary languages. Then, the paper presented a formal language that fulfills these requirements, encoded a single-commitment bit protocol in that language, and proved that the protocol is UC by proving that the compiler generating that protocol attains *RHP*. Finally, the paper mechanised the *RHP* proof in Deepsec for both the static and the dynamic corruption cases, providing a scalable, mechanised proof of UC via our connection.

ACKNOWLEDGMENTS

The authors would like to thank Carmine Abate, Amal Ahmed, Dan Boneh, Deepak Garg, John Mitchell, Jeremy Thibault for useful feedback and discussions. This work was partially supported by the Office of Naval Research for support through grant N00014-18-1-2620, Accountable Protocol Customization; the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity (FKZ: 13N1S0762); the Italian Ministry of Education through funding for the Rita Levi Montalcini grant (call of 2019).

REFERENCES

[1] Martín Abadi. 1998. Protection in Programming-Language Translations. In *ICALP'98*. 868–883.

[2] Martín Abadi, Bowen Alpern, Krzysztof R. Apt, Nissim Francez, Shmuel Katz, Leslie Lamport, and Fred B. Schneider. 1991. Preserving Liveness: Comments on "Safety and Liveness from a Methodological Point of View". *Inf. Process. Lett.* 40, 3 (1991), 141–142. DOI:http://dx.doi.org/10.1016/0020-0190(91)90168-H

[3] Martín Abadi and Véronique Cortier. 2006. Deciding Knowledge in Security Protocols under Equational Theories. *Theoretical Computer Science* 367, 1-2 (Nov. 2006), 2–32. DOI:http://dx.doi.org/10.1016/j.tcs.2006.08.032

[4] Martín Abadi and Cédric Fournet. 2001. Mobile Values, New Names, and Secure Communication. In *POPL*. ACM, 104–115.

[5] Martín Abadi and Cédric Fournet. 2001. Mobile Values, New Names, and Secure Communication. *SIGPLAN Not.* 36, 3 (Jan. 2001), 104–115. DOI:http://dx.doi.org/10.1145/373243.360213

[6] Martín Abadi, Cédric Fournet, and Georges Gonthier. 1998. Secure Implementation of Channel Abstractions. In *IEEE Symposium on Logic in Computer Science*. 105–116.

[7] Martín Abadi, Cédric Fournet, and Georges Gonthier. 2000. Authentication Primitives and their Compilation. In *Principles of Programming Languages*. ACM, 302–315.

[8] Martín Abadi, Cédric Fournet, and Georges Gonthier. 2002. Secure Implementation of Channel Abstractions. *Information and Computation* 174, 1 (2002), 37–83. DOI:http://dx.doi.org/10.1006/inco.2002.3086

[9] Martín Abadi and Andrew D. Gordon. 1999. A Calculus for Cryptographic Protocols: The spi Calculus. *Inf. Comput.* 148, 1 (1999), 1–70.

[10] Martín Abadi and Gordon D. Plotkin. 2012. On Protection by Layout Randomization. *ACM Transactions on Information and System Security* 15, Article 8 (July 2012), 8:1–8:29 pages.

[11] Carmine Abate, Arthur Azevedo de Amorim, Roberto Blanco, Ana Nora Evans, Guglielmo Fachini, Catalin Hritcu, Théo Laurent, Benjamin C. Pierce, Marco Stronati, and Andrew Tolmach. 2018. When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise *(CCS '18)*.

[12] Carmine Abate, Roberto Blanco, Ştefan Ciobâcă, Adrien Durier, Deepak Garg, Catalin Hritcu, Marco Patrignani, Éric Tanter, and Jeremy Jérémy. 2020. Trace-Relating Compiler Correctness and Secure Compilation. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020 (ESOP)*. 1–28. DOI:http://dx.doi.org/10.1007/978-3-030-44914-8_1

[13] Carmine Abate, Roberto Blanco, Ştefan Ciobâcă, Adrien Durier, Deepak Garg, Cătălin Hriţcu, Marco Patrignani, Éric Tanter, and Jérémy Thibault. 2021. An Extended Account of Trace-Relating Compiler Correctness and Secure Compilation. *ACM Trans. Program. Lang. Syst.* 43, 4, Article 14 (nov 2021), 48 pages. DOI:http://dx.doi.org/10.1145/3460860

[14] Carmine Abate, Roberto Blanco, Deepak Garg, Cătălin Hriţcu, Marco Patrignani, and Jérémy Thibault. 2019. Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation. In *2019 IEEE 32th Computer Security Foundations Symposium (CSF 2019)*.

[15] Carmine Abate, Matteo Busi, and Stelios Tsampas. 2021. Fully Abstract and Robust Compilation: And How to Reconcile the Two, Abstractly. In *Programming Languages and Systems - 19th Asian Symposium, APLAS 2021, Chicago, IL, USA, October 17-18, 2021, Proceedings (Lecture Notes in Computer Science)*, Hakjoo Oh (Ed.), Vol. 13008. Springer, 83–101. DOI:http://dx.doi.org/10.1007/978-3-030-89051-3_6

[16] Michel Abdalla, Manuel Barbosa, Tatiana Bradley, Stanisław Jarecki, Jonathan Katz, and Jiayu Xu. 2020. Universally Composable Relaxed Password Authenticated Key Exchange. In *Advances in Cryptology – CRYPTO 2020 (Lecture Notes in Computer Science)*, Daniele Micciancio and Thomas Ristenpart (Eds.). Springer International Publishing, Cham, 278–307. DOI:http://dx.doi.org/10.1007/978-3-030-56784-2_10

[17] Coşku Acay, Rolph Recto, Joshua Gancher, Andrew C. Myers, and Elaine Shi. 2021. Viaduct: An Extensible, Optimizing Compiler for Secure Distributed Programs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 740–755. DOI:http://dx.doi.org/10.1145/3453483.3454074

[18] Luca Aceto, Anna Ingólfsdóttir, Kim Guldstrand Larsen, and Jiri Srba. 2007. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press.

[19] Amal Ahmed and Matthias Blume. 2008. Typed Closure Conversion Preserves Observational Equivalence. In *International Conference on Functional Programming*. ACM, 157–168.

[20] Amal Ahmed and Matthias Blume. 2011. An Equivalence-Preserving CPS Translation via Multi-Language Semantics. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11)*. ACM, 431–444.

[21] Mihhail Aizatulin, Andrew D. Gordon, and Jan Jürjens. 2011. Extracting and Verifying Cryptographic Models from C Protocol Code by Symbolic Execution. In *Proceedings of the 18th ACM Conference on Computer and Communications Security - CCS '11*. ACM Press, Chicago, Illinois, USA, 331. DOI:http://dx.doi.org/10.1145/2046707.2046745

[22] Michael Backes, Robert Künnemann, and Esfandiar Mohammadi. 2016. Computational Soundness for Dalvik Bytecode. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. Association for Computing Machinery, New York, NY, USA, 717–730. DOI:http://dx.doi.org/10.1145/2976749.2978418

[23] Michael Backes, Birgit Pfitzmann, and Michael Waidner. 2004. Secure Asynchronous Reactive Systems. (04 2004). Technical report 082.

[24] Michael Backes, Birgit Pfitzmann, and Michael Waidner. 2007. The reactive simulatability (RSIM) framework for asynchronous systems. *Inf. Comput.* 205, 12 (2007), 1685–1720. DOI:http://dx.doi.org/10.1016/j.ic.2007.05.002

[25] David Baelde, Stéphanie Delaune, Charlie Jacomme, Adrien Koutsos, and Solène Moreau. 2021. An Interactive Prover for Protocol Verification in the Computational Model. In *SP 2021 - 42nd IEEE Symposium on Security and Privacy (Proceedings of the 42nd IEEE Symposium on Security and Privacy (s &P'21))*. San Fransisco / Virtual, United States.

[26] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. 2021. SoK: Computer-Aided Cryptography. In *2021 IEEE Symposium on Security and Privacy (SP)*. 777–795. DOI: http://dx.doi.org/10.1109/SP40001.2021.00008

[27] Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, and Pierre-Yves Strub. 2021. Mechanized Proofs of Adversarial Complexity and Application to Universal Composability. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Virtual Event Republic of Korea, 2541–2563. DOI: http://dx.doi.org/10.1145/3460120.3484548

[28] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. 2011. Computer-Aided Security Proofs for the Working Cryptographer. In *Advances in Cryptology – CRYPTO 2011*, Phillip Rogaway (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 71–90.

[29] Gilles Barthe, Benjamin Grégoire, Charlie Jacomme, Steve Kremer, and Pierre-Yves Strub. 2019. Symbolic Methods in Computational Cryptography Proofs. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. 136–13615. DOI:http://dx.doi.org/10.1109/CSF.2019.00017

[30] David Basin, Andreas Lochbihler, Ueli Maurer, and S. Reza Sefidgar. 2021. Abstract Modeling of System Communication in Constructive Cryptography Using CryptHOL. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. 1–16. DOI:http://dx.doi.org/10.1109/CSF51468.2021.00047

[31] Mihir Bellare and Phillip Rogaway. 2004. *The Game-Playing Technique*. Technical Report.

[32] B. Blanchet. 2008. A Computationally Sound Mechanized Prover for Security Protocols. *IEEE Trans. Dependable and Secure Comput.* 5, 4 (Oct. 2008), 193–207. DOI:http://dx.doi.org/10.1109/TDSC.2007.1005

[33] Florian Böhl and Dominique Unruh. 2013. Symbolic Universal Composability. In *2013 IEEE 26th Computer Security Foundations Symposium*. 257–271. DOI:http://dx.doi.org/10.1109/CSF.2013.24

[34] William J. Bowman and Amal Ahmed. 2015. Noninterference for free. In *ICFP*. ACM.

[35] Matteo Busi, Pierpaolo Degano, and Letterio Galletta. 2022. Towards Effective Preservation of Robust Safety Properties. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing (SAC '22)*. Association for Computing Machinery, New York, NY, USA, 1674–1683. DOI:http://dx.doi.org/10.1145/3477314.3507084

[36] Jan Camenisch, Stephan Krenn, Ralf Küsters, and Daniel Rausch. 2019. iUC: Flexible Universal Composability Made Simple. In *Advances in Cryptology - ASIACRYPT 2019 - 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8-12, 2019, Proceedings, Part III (Lecture Notes in Computer Science)*, Vol. 11923. Springer, 191–221.

[37] R. Canetti. 2001. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. IEEE, Newport Beach, CA, USA, 136–145. DOI:http://dx.doi.org/10.1109/SFCS.2001.959888

[38] Ran Canetti. 2001. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *Proceedings of the 42Nd IEEE Symposium on Foundations of Computer Science (FOCS '01)*. IEEE Computer Society, Washington, DC, USA, 136–. http://dl.acm.org/citation.cfm?id=874063.875553

[39] Ran Canetti. 2020. Universally Composable Security. *J. ACM* 67, 5 (Sept. 2020), 28:1–28:94. DOI:http://dx.doi.org/10.1145/3402457

[40] Ran Canetti, Asaf Cohen, and Yehuda Lindell. 2014. A Simpler Variant of Universally Composable Security for Standard Multiparty Computation. Cryptology ePrint Archive, Report 2014/553. (2014). https://eprint.iacr.org/2014/553.

[41] Ran Canetti and Marc Fischlin. 2001. Universally Composable Commitments. In *Advances in Cryptology — CRYPTO 2001*, Joe Kilian (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 19–40.

[42] Ran Canetti, Palak Jain, Marika Swanberg, and Mayank Varia. Universally Composable End-to-End Secure Messaging: A Modular Analysis. In *CRYPTO'2022*. 78.

[43] Ran Canetti, Alley Stoughton, and Mayank Varia. 2019. EasyUC: Using EasyCrypt to Mechanize Proofs of Universally Composable Security. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. 167–16716. DOI:http://dx.doi.org/10.1109/CSF.2019.00019

[44] Vincent Cheval, Steve Kremer, and Itsaka Rakotonirina. 2018. DEEPSEC: Deciding Equivalence Properties in Security Protocols Theory and Practice. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, 529–546. DOI:http://dx.doi.org/10.1109/SP.2018.00033

[45] Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *Journal of Computer Security* 18, 6 (2010), 1157–1210. DOI:http://dx.doi.org/10.3233/JCS-2009-0393

[46] Anupam Datta, Ralf Küsters, John C. Mitchell, and Ajith Ramanathan. 2005. On the Relationships Between Notions of Simulation-based Security. In *Proceedings of the Second International Conference on Theory of Cryptography (TCC'05)*. Springer-Verlag, 476–494. DOI:http://dx.doi.org/10.1007/978-3-540-30576-7_26

[47] Stéphanie Delaune and Lucca Hirschi. 2017. A Survey of Symbolic Methods for Establishing Equivalence-Based Properties in Cryptographic Protocols. *Journal of Logical and Algebraic Methods in Programming* 87 (Feb. 2017), 127–144. DOI:http://dx.doi.org/10.1016/j.jlamp.2016.10.005

[48] Stéphanie Delaune, Steve Kremer, and Olivier Pereira. 2009. Simulation Based Security in the Applied Pi Calculus. *Cryptology ePrint Archive* (2009).

[49] Dominique Devriese, Marco Patrignani, Frank Piessens, and Steven Keuchel. 2017. Modular, Fully-abstract Compilation by Approximate Back-translation. *Logical Methods in Computer Science* Volume 13, Issue 4 (Oct. 2017).

[50] Akram El-Korashy, Roberto Blanco, Jérémy Thibault, Adrien Durier, Deepak Garg, and Catalin Hritcu. 2022. SecurePtrs: Proving Secure Compilation with Data-Flow Back-Translation and Turn-Taking Simulation. (2022). arXiv:cs.PL/2110.01439

[51] Akram El-Korashy, Stelios Tsampas, Marco Patrignani, Dominique Devriese, Deepak Garg, and Frank Piessens. 2021. CapablePtrs: Securely Compiling Partial Programs Using the Pointers-as-Capabilities Principle. In *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021*. 1–16. DOI:http://dx.doi.org/10.1109/CSF51468.2021.00036

[52] Cédric Fournet, Markulf Kohlweiss, and Pierre-Yves Strub. 2011. Modular Code-Based Cryptographic Verification. In *Proceedings of the 18th ACM Conference on Computer and Communications Security - CCS '11*. ACM Press, Chicago, Illinois, USA, 341. DOI:http://dx.doi.org/10.1145/2046707.2046746

[53] Cédric Fournet, Markulf Kohlweiss, and Pierre-Yves Strub. 2011. Modular Code-Based Cryptographic Verification. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11)*. Association for Computing Machinery, New York, NY, USA, 341–350. DOI:http://dx.doi.org/10.1145/2046707.2046746

[54] Oded Goldreich. 1990. A Note on Computational Indistinguishability. *Inf. Process. Lett.* 34, 6 (1990), 277–281. DOI:http://dx.doi.org/10.1016/0020-0190(90)90010-U

[55] Roberto Guanciale, Musard Balliu, and Mads Dam. 2020. InSpectre: Breaking and Fixing Microarchitectural Vulnerabilities by Formal Analysis. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Virtual Event USA, 1853–1869. DOI:http://dx.doi.org/10.1145/3372297.3417246

[56] H. Haagh, A. Karbyshev, S. Oechsner, B. Spitters, and P. Strub. 2018. Computer-Aided Proofs for Multiparty Computation with Active Security. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE Computer Society, Los Alamitos, CA, USA, 119–131. DOI:http://dx.doi.org/10.1109/CSF.2018.00016

[57] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. 2019. SoK: General Purpose Compilers for Secure Multi-Party Computation. In *2019 IEEE Symposium on Security and Privacy (SP)*. 1220–1237. DOI:http://dx.doi.org/10.1109/SP.2019.00028

[58] Michael Hicks. 2014. Formal Reasoning in PL and Crypto. (2014). Available at: http://www.pl-enthusiast.net/2014/12/23/formal-reasoning-pl-crypto/.

[59] Martin Hirt, Chen-Da Liu-Zhang, and Ueli Maurer. 2021. Adaptive Security of Multi-Party Protocols, Revisited. In *Theory of Cryptography Conference*.

[60] Dennis Hofheinz and Victor Shoup. 2011. GNUC: A New Universal Composability Framework. Cryptology ePrint Archive. (2011). http://eprint.iacr.org/

[61] Dennis Hofheinz and Dominique Unruh. 2006. Simulatable security and polynomially bounded concurrent composability. In *IEEE Symposium on Security and Privacy*.

[62] Dennis Hofheinz, Dominique Unruh, and Jörn Müller-Quade. 2013. Polynomial runtime and composability. *Journal of Cryptology* 26, 3 (2013), 375–441.

[63] Dennis Hofheinz, Dominique Unruh, and Jörn Müller-Quade. 2013. Polynomial Runtime and Composability. *J Cryptol* 26, 3 (July 2013), 375–441. DOI:http://dx.doi.org/10.1007/s00145-012-9127-4

[64] Chung-Kil Hur and Derek Dreyer. 2011. A Kripke Logical Relation Between ML and Assembly. In *Principles of Programming Languages*. 133–146.

[65] Alan Jeffrey and Julian Rathke. 2005. Java Jr.: Fully abstract trace semantics for a core Java language. In *ESOP'05 (LNCS)*, Vol. 3444. Springer, 423–438.

[66] Matthis Kruse and Marco Patrignani. 2022. Composing Secure Compilers.

[67] Ralf Kuesters, Max Tuengerthal, and Daniel Rausch. 2013. *The IITM Model: A Simple and Expressive Model for Universal Composability.* Technical Report 025.

[68] Robert Künnemann, Marco Patrignani, and Ethan Cecchetti. Computationally Bounded Robust Compilation and Universally Composable Security. In *CSF 2024.*

[69] Ralf Küsters. 2006. Simulation-Based Security with Inexhaustible Interactive Turing Machines. In *Computer Security Foundations Workshop.* IEEE Computer Society, 309–320.

[70] Ralf Kusters, Tomasz Truderung, Bernhard Beckert, Daniel Bruns, Michael Kirsten, and Martin Mohr. 2015. A Hybrid Approach for Proving Noninterference of Java Programs. In *2015 IEEE 28th Computer Security Foundations Symposium.* IEEE, Verona, 305–319. DOI:http://dx.doi.org/10.1109/CSF.2015.28

[71] Leslie Lamport and Fred B. Schneider. 1984. Formal Foundation for Specification and Verification. In *Distributed Systems: Methods and Tools for Specification, An Advanced Course.* 203–285. DOI:http://dx.doi.org/10.1007/3-540-15216-4_15

[72] Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *Journal of Automated Reasoning* 43, 4 (2009), 363–446. http://dx.doi.org/10.1007/s10817-009-9155-4

[73] Kevin Liao, Matthew A. Hammer, and Andrew Miller. 2019. ILC: A Calculus for Composable, Computational Cryptography. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019).* ACM, 640–654. DOI:http://dx.doi.org/10.1145/3314221.3314607

[74] Jacob Matthews and Robert Bruce Findler. 2009. Operational Semantics for Multi-language Programs. *ACM Trans. Program. Lang. Syst.* 31, 3 (April 2009), 12:1–12:44.

[75] Ueli Maurer. 2011. Constructive Cryptography - A New Paradigm for Security Definitions and Proofs. In *Theory of Security and Applications - Joint Workshop, TOSCA 2011, Saarbrücken, Germany, March 31 - April 1, 2011, Revised Selected Papers (Lecture Notes in Computer Science)*, Sebastian Mödersheim and Catuscia Palamidessi (Eds.), Vol. 6993. Springer, 33–56. DOI:http://dx.doi.org/10.1007/978-3-642-27375-9_3

[76] John C. Mitchell, Rahul Sharma, Deian Stefan, and Joe Zimmerman. 2012. Information-Flow Control for Programming on Encrypted Data. In *2012 IEEE 25th Computer Security Foundations Symposium.* 45–60. DOI:http://dx.doi.org/10.1109/CSF.2012.30

[77] Greg Morrisett, Elaine Shi, Kristina Sojakova, Xiong Fan, and Joshua Gancher. 2021. IPDL: A Simple Framework for Formally Verifying Distributed Cryptographic Protocols. Cryptology ePrint Archive, Paper 2021/147. (2021). https://eprint.iacr.org/2021/147 https://eprint.iacr.org/2021/147.

[78] Marco Patrignani. 2020. Why Should Anyone use Colours? or, Syntax Highlighting Beyond Code Snippets. (2020). arXiv:cs.SE/2001.11334

[79] Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. 2015. Secure Compilation to Protected Module Architectures. *ACM Trans. Program. Lang. Syst.* 37, Article 6 (April 2015), 6:1–6:50 pages.

[80] Marco Patrignani, Amal Ahmed, and Dave Clarke. 2019. Formal Approaches to Secure Compilation A Survey of Fully Abstract Compilation and Related Work. *ACM Comput. Surv.* 51, 6, Article 125 (Jan. 2019), 36 pages.

[81] Marco Patrignani and Dave Clarke. 2015. Fully abstract trace semantics for protected module architectures. *Elsevier COMLAN* 42, 0 (2015), 22 – 45. DOI:http://dx.doi.org/10.1016/j.cl.2015.03.002

[82] Marco Patrignani, Dominique Devriese, and Frank Piessens. 2016. On Modular and Fully Abstract Compilation. In *Computer Security Foundations Symposium.*

[83] Marco Patrignani and Deepak Garg. 2019. Robustly Safe Compilation. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019 (ESOP'19).*

[84] Marco Patrignani and Deepak Garg. 2021. Robustly Safe Compilation, an Efficient Form of Secure Compilation. *ACM Trans. Program. Lang. Syst.* 43, 1, Article 1 (feb 2021), 41 pages. DOI:http://dx.doi.org/10.1145/3436809

[85] Marco Patrignani and Marco Guarnieri. 2021. Exorcising Spectres with Secure Compilers. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21).* Association for Computing Machinery, New York, NY, USA, 445–461. DOI:http://dx.doi.org/10.1145/3460120.3484534

[86] Marco Patrignani, Riad S. Wahby, and Robert Künnemann. 2019. Universal Composability is Secure Compilation. (2019). arXiv:1910.08634

[87] James T. Perconti and Amal Ahmed. 2014. Verifying an Open Compiler Using Multi-language Semantics. In *ESOP (LNCS)*, Vol. 8410. 128–148.

[88] Benjamin Pierce and Eijiro Sumii. 2000. Relating Cryptography and Polymorphism. (2000). http://www.kb.ecei.tohoku.ac.jp/~sumii/pub/infohide.pdf manuscript.

[89] Gordon D. Plotkin. 1977. LCF Considered as a Programming Language. *Theoretical Computer Science* 5 (1977), 223–255.

[90] Daniel Rausch, Ralf Küsters, and Céline Chevalier. 2022. Embedding the UC Model into the IITM Model. In *Advances in Cryptology – EUROCRYPT 2022 (Lecture Notes in Computer Science)*, Orr Dunkelman and Stefan Dziembowski (Eds.). Springer International Publishing, Cham, 242–272. DOI:http://dx.doi.org/10.1007/978-3-031-07085-3_9

[91] Davide Sangiorgi and David Walker. 2001. *PI-Calculus: A Theory of Mobile Processes.* Cambridge University Press, New York, NY, USA.

[92] Fred B. Schneider. 2000. Enforceable security policies. *ACM Transactions of Information Systems Security* 3, 1 (2000), 30–50. http://www.cs.cornell.edu/fbs/publications/EnfSecPols.pdf

[93] Eijiro Sumii and Benjamin C. Pierce. 2003. Logical Relations for Encryption. *J. Comput. Secur.* 11, 4 (July 2003), 521–554.

[94] Eijiro Sumii and Benjamin C. Pierce. 2004. A Bisimulation for Dynamic Sealing. In *Principles of Programming Languages*. 161–172.

[95] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. Association for Computing Machinery, New York, NY, USA, 256–270. DOI:http://dx.doi.org/10.1145/2837614.2837655