

# SSProve: A foundational framework for modular cryptographic proofs in Coq

CARMINE ABATE, MPI-SP, Germany

PHILIPP G. HASELWARTER, Aarhus University, Denmark

EXEQUIEL RIVAS, Tallinn University of Technology, Estonia and LigoLANG, France

ANTOINE VAN MUYLDER, KU Leuven, Belgium

THÉO WINTERHALTER, MPI-SP, Germany

NIKOLAJ SIDORENCO, Aarhus University, Denmark

CĂTĂLIN HRIȚCU, MPI-SP, Germany

KENJI MAILLARD, Inria Rennes, France

BAS SPITTERS, Aarhus University, Denmark

State-separating proofs (SSP) is a recent methodology for structuring game-based cryptographic proofs in a modular way. While very promising, this methodology was previously not fully formalized and came with little tool support. We address this by introducing SSProve, the first general verification framework for machine-checked state-separating proofs. SSProve combines high-level modular proofs about composed protocols, as proposed in SSP, with a probabilistic relational program logic for formalizing the lower-level details, which together enable constructing fully machine-checked cryptographic proofs in the Coq proof assistant. Moreover, SSProve is itself formalized in Coq, including the algebraic laws of SSP, the soundness of the program logic, and the connection between these two verification styles. To illustrate the formal SSP methodology we prove security of ElGamal and PRF-based encryption. We also validate the SSProve approach by conducting two extended case studies. First, we formalize a security proof of the KEM-DEM public key encryption scheme. Second, we formalize security of the sigma-protocol zero-knowledge construction and the associated construction of commitment schemes. We then instantiate the proof and give concrete security bounds for Schnorr's protocol.

CCS Concepts: • **Theory of computation** → *Logic and verification*; **Programming logic**; *Categorical semantics*; Invariants; Pre- and post-conditions; *Program verification*; *Probabilistic computation*; • **Security and privacy** → **Cryptography**; **Formal methods and theory of security**; **Logic and verification**;

Additional Key Words and Phrases: software verification, probabilistic relational program logic, state-separating proofs, game-playing cryptographic proofs, formal methods

## ACM Reference Format:

Carmine Abate, Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Nikolaj Sidorenco, Cătălin Hrițcu, Kenji Maillard, and Bas Spitters. 2021. SSProve: A foundational framework for modular cryptographic proofs in Coq. 1, 1 (October 2021), 54 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

Authors' addresses: Carmine Abate, [carmine.abate@mpi-sp.org](mailto:carmine.abate@mpi-sp.org), MPI-SP, Bochum, Germany; Philipp G. Haselwarter, [philipp@haselwarter.org](mailto:philipp@haselwarter.org), Aarhus University, Aarhus, Denmark; Exequiel Rivas, [erivas@dcc.fceia.unr.edu.ar](mailto:erivas@dcc.fceia.unr.edu.ar), Tallinn University of Technology, Tallinn, Estonia and LigoLANG, Paris, France; Antoine Van Muylder, [antoine.vanmuylder@kuleuven.be](mailto:antoine.vanmuylder@kuleuven.be), KU Leuven, Brussels, Belgium; Théo Winterhalter, [theo.winterhalter@mpi-sp.org](mailto:theo.winterhalter@mpi-sp.org), MPI-SP, Bochum, Germany; Nikolaj Sidorenco, [sidorenco@cs.au.dk](mailto:sidorenco@cs.au.dk), Aarhus University, Aarhus, Denmark; Cătălin Hrițcu, [catalin.hritcu@mpi-sp.org](mailto:catalin.hritcu@mpi-sp.org), MPI-SP, Bochum, Germany; Kenji Maillard, [kenji.maillard@inria.fr](mailto:kenji.maillard@inria.fr), Inria Rennes, Nantes, France; Bas Spitters, [spitters@cs.au.dk](mailto:spitters@cs.au.dk), Aarhus University, Aarhus, Denmark.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2021 Copyright held by the owner/author(s).

XXXX-XXXX/2021/10-ART

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

## 1 INTRODUCTION

Cryptographic proofs can be challenging to make fully precise and to rigorously check. This has caused a “crisis of rigor” [19] in cryptography that Shoup [59], Bellare and Rogaway [19], Halevi [36], and others, proposed to address by systematically structuring proofs as sequences of games. This game-based proof methodology is not only ubiquitous in provable cryptography nowadays, but also amenable to full machine-checking in proof assistants such as Coq [16, 51] and Isabelle/HOL [18]. It has also led to the development of specialized proof assistants [12] and automated verification tools for cryptographic proofs [11, 15, 24]. There are two key ideas behind these tools: (i) formally representing games and the adversaries against them as code in a probabilistic programming language, and (ii) using program verification techniques to conduct all game transformation steps in a machine-checked manner.

For a long time however, game-based proofs have lacked modularity, which made them hard to scale to large, composed protocols such as TLS [55] or the upcoming MLS [10]. To address this issue, Brzuska et al. [25] have recently introduced *state-separating proofs (SSP)*, a methodology for modular game-based proofs, inspired by the paper proofs in the miTLS project [22, 23, 34], by prior compositional cryptography frameworks [27, 47], and by process algebras [48]. In the SSP methodology, the code of cryptographic games is split into packages, which are modules made up of procedures sharing state. Packages can call each other’s procedures (also known as oracles) and can operate on their own state, but cannot directly access other packages’ state. Packages have natural notions of sequential and parallel composition that satisfy simple algebraic laws, such as associativity of sequential composition. This law is used to define cryptographic reductions not only in SSP, but also in the *The Joy of Cryptography* textbook [57], which teaches cryptographic proofs in a style very similar to SSP.

While the SSP methodology is very promising, the lack of a complete formalization makes it currently only usable for informal paper proofs, not for machine-checked ones. The SSP paper [25] defines package composition and the syntax of a cryptographic pseudocode language for games and adversaries, but the semantics of this language is not formally defined, and the meaning of their `assert` operator is not explained and not self-evident, given the probabilistic setting. Moreover, while SSP provides a good way to structure proofs at the high-level, using algebraic laws such as associativity, the low-level details of such proofs are usually treated very casually on paper. Yet none of the existing cryptographic verification tools that could help machine-check these low-level details supports the high-level part of SSP proofs: equational reasoning about composed packages (i.e., modules) is either not possible at all [16, 36, 51, 63], or does not exactly match the SSP package abstraction [12, 42] (see §8 for details).

The main contribution of this work is to introduce SSProve, the first general verification framework for machine-checked state-separating proofs. SSProve brings together two different proof styles into a single unified framework: (1) high-level proofs are modular, done by reasoning equationally about composed packages, as proposed in SSP [25]; (2) low-level details are formally proved in a probabilistic relational program logic [12, 16, 51]. Importantly, we show a formal connection between these two proof styles in Theorem 2.4.

SSPprove is a foundational framework, fully formalized in Coq. To achieve this, we define the syntax of cryptographic pseudocode in terms of a free monad, in which external calls are represented as algebraic operations [52]. This gives us a principled way to define sequential composition of packages based on an algebraic effect handler [53] and to give machine-checked proofs of the SSP package laws [25], some of which were treated informally on paper. Moreover, we make precise the minimal state-separation requirements between adversaries and the games with which they are

composed—this reduces the proof burden and allows us to prove more meaningful security results, that do not require the adversary’s state to be disjoint from intermediate games in the proof.

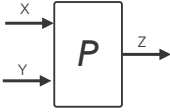
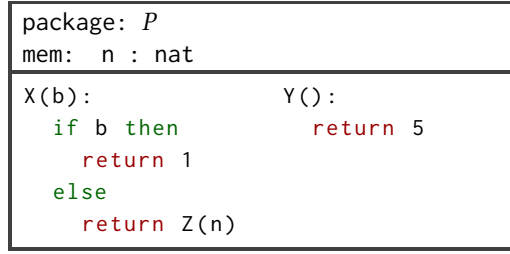
Beyond syntax, we also give a denotational semantics to cryptographic code in terms of stateful probabilistic functions that can signal assertion failures by sampling from the empty probability subdistribution. Finally, we prove the soundness of a probabilistic relational program logic for deriving properties about pairs of cryptographic code fragments.

For this soundness proof we build a semantic model based on relational weakest-precondition specifications. Our model is modular with respect to the considered side-effects (currently probabilities, state, and assertion failures). To obtain it, we follow a general recipe by Maillard et al. [45], who recently proposed to characterize such semantic models as relative monad morphisms, mapping two monadic computations to their canonical relational specification. This allows us to first define a relative monad morphism for probabilistic, potentially failing computations and then to extend this to state by applying a relative monad transformer. Working out this instance of Maillard et al.’s [45] recipe involved formalizing various non-standard categorical constructs in Coq, in an order-enriched context: lax functors, lax natural transformations, left relative adjunctions, lax morphisms between such adjunctions, state transformations of such adjunctions, etc. This formalization is of independent interest and should also allow us to more easily add extra side-effects and  $F^*$ -style sub-effecting [63] to SSProve in the future.

We put SSProve to the test by formalizing several security proofs. Security of PRF-based encryption, which was also presented in [25], and security of ElGamal public key encryption is discussed in §2.3 and §2.4 respectively. We have additionally formalised two case studies to validate our approach beyond the scope of these illustrative examples. We formalised the security of the KEM-DEM public key encryption scheme of Cramer and Shoup [30], which originally served to illustrate SSP ideas in [25]. The proof extensively uses the package laws of SSP and showcases formal reasoning with invariants. Furthermore, we give a new proof of security of  $\Sigma$ -protocols in SSP style, and show how any  $\Sigma$ -protocol can be used to construct a commitment scheme. We define a concrete example of a  $\Sigma$ -protocol following Schnorr [58] and prove concrete security bounds.

We have already started to reap the benefits of formalizing SSP in a proof assistant: our formalization of the KEM-DEM case study presented in [25] has led us to find—in conjunction with the authors of [25]—an error in the originally published proof. The authors of [25] have since proposed a revised version of their theorem, which we have adapted and fully proved in SSProve. In turn, Markulf Kohlweiss has alerted us about a weakness in the security definition of public-key encryption schemes used in the conference version of the current paper [1]. We were able to quickly fix the security definition and formalised proof of ElGamal, as discussed in §2.4. This demonstrates that the language of SSProve is comprehensible to independent experts, who can review security definitions. An advantage of embedding SSProve into the Coq proof assistant is that we can readily reuse our security definitions across developments, thus increasing confidence in the correctness not only of our proofs but also of our definitions.

*Outline.* The remainder of this paper is structured as follows. §2 illustrates the key ideas of how to use SSProve on two simple cryptographic proofs, showing semantic security of ElGamal and PRF-based encryption. In §3 we formalize the SSP methodology: cryptographic pseudocode, packages, sequential and parallel composition, and the algebraic laws they satisfy. In §4 we introduce the rules of our probabilistic relational program logic and use them to prove Theorem 2.4, which formally connects SSP to this program logic. In §5 we outline the effect-modular semantic model we use to prove the soundness of the program logic. In §6 we present an extended case study, formalizing security of the KEM-DEM public key encryption scheme of Cramer and Shoup [30], following the proof given in [25]. In §7 we present a novel formalisation of  $\Sigma$ -protocols in SSProve as a second case study. Finally, §8 discusses related work and §9 future directions.

Fig. 1. Package  $P$ Fig. 2. Possible pseudocode implementation for  $P$ 

The full formalization of SSProve and of the examples from this paper (circa 24K lines of Coq code including comments) are available under the MIT open source license at <https://github.com/SSProve/ssprove/releases/tag/journal-submission>.

*Remark.* A previous version [1] of the present paper has been published at CSF 2021. The improvements we made throughout the text are too many to list exhaustively. At a high level we: (i) corrected the ElGamal security definition in §2.4; (ii) expanded the explanation of the logical rules in §4 and added new rules for assertions, asynchronous memory accesses, and state invariants; (iii) significantly expanded the section on semantics (§5) to be more self-contained and accessible, and to draw connections to related approaches; (iv) added Sections 6 and 7.

## 2 USING SS Prove: KEY IDEAS AND EXAMPLES

Formalizing the SSP methodology for high-level proofs allows us to formally link it to the methodology of probabilistic relational program logics for low-level proofs. In this section, we begin with a brief introduction to SSP (§2.1). Then, we present our new theorem connecting SSP to a probabilistic relational program logic (§2.2). Finally, by way of two examples, we show how the two methodologies are used together to obtain fully formal security proofs. The first example looks at a symmetric encryption scheme built out of a pseudo-random function (§2.3), while the second looks at ElGamal, a popular asymmetric encryption scheme (§2.4).

### 2.1 An introduction to SSP

We begin by introducing (our variant of) the SSP methodology of Brzuska et al. [25]. The main concept behind this methodology is the *package*, which is a collection of procedure implementations that together manipulate a common piece of state, and that may depend on a set of external procedures. We refer to the set of external procedures on which the package can depend as the *imports* of the package. In Figure 1, we can see a high-level picture of a package  $P$ : it implements and *exports* the procedures  $X$  and  $Y$ , and it imports the external procedure  $Z$ . The arrows indicate the direction of calls, i.e. exports that can be called from the outside point towards  $P$  and imports point away. We use  $\text{import}(P)$  to denote the set of procedure names the package  $P$  imports, and  $\text{export}(P)$  to denote the names of the procedures it exports. The term *interface* is used to refer to such a set of procedure names.<sup>1</sup> While the import and export interfaces of a package tell us where it can be used, in the SSP papers, the package implementations are usually given in separate figures, which describe, in pseudocode, each of the procedures exported by the package. For example, a possible pseudocode implementation corresponding to the package  $P$  can be found in Figure 2. We refer to the code of the procedure  $X$  exported by package  $P$  as  $P.X$ .

<sup>1</sup>In SSProve the procedure names within interfaces are also associated with argument and result types, but we omit this detail until §3.1.

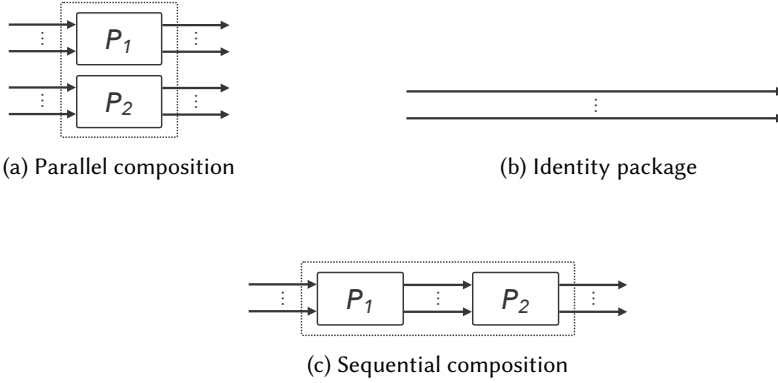


Fig. 3. Graphical representation of packages

*Package algebra.* Packages can be combined as algebraic objects. We can build complex packages out of simpler ones using the following composition operations:<sup>2</sup>

- *Sequential composition:* given two packages  $P_1$  and  $P_2$  with  $\text{import}(P_1) \subseteq \text{export}(P_2)$ , then  $P_1 \circ P_2$  is obtained by inlining procedure definitions, each time  $P_1$  calls a procedure in  $P_2$ .
- *Parallel composition:* given two packages  $P_1$  and  $P_2$  such that  $\text{export}(P_1)$  and  $\text{export}(P_2)$  are disjoint, then  $P_1 \parallel P_2$  is the union of  $P_1$  and  $P_2$ : it provides the procedures from both  $P_1$  and  $P_2$ .
- *Identity package:* given an interface  $I$ , we have a package that simply forwards all calls in this interface. We refer to it as the identity package on the interface  $I$ , written  $\text{ID}_I$ , and we have that  $\text{import}(\text{ID}_I) = \text{export}(\text{ID}_I) = I$ .

These operations have graphical counterparts which we show in Figure 3: parallel composition (Figure 3a) is represented by stacking packages on top of each other; sequential composition (Figure 3c) is obtained by merging the input arrows of one of the packages with the output arrows of the other; finally the identity package (Figure 3b) is essentially silent when represented graphically, its presence being notified by longer arrows. Moreover, there are natural algebraic laws that hold between these operators. For example, sequential composition is an associative operator, which formally we can state by the following equation:

$$P_1 \circ (P_2 \circ P_3) \stackrel{\text{code}}{\equiv} (P_1 \circ P_2) \circ P_3 \quad (1)$$

Graphically these laws are obtained by simply forgetting about the dashed boxes (which represent parenthesizing) and by stretching arrows. In the SSP methodology, the  $\stackrel{\text{code}}{\equiv}$  symbol stands for code equality between the packages: two packages are equal if the implementations of their procedures are equal to each other. As code equality corresponds precisely to syntactic equality in our representation, we will write  $P = Q$  instead of  $P \stackrel{\text{code}}{\equiv} Q$  in the remainder of the paper. The aforementioned algebraic package laws (see subsection 3.4 for details) are convenient for cryptographic proofs, since they allow the compositional structure of a package to be manipulated without having to look at all at the implementation of its procedures.

*Games and distinguishers.* A package with no imports is called a *game*. A game pair contains two games that export the same procedures, i.e. a tuple  $(G^0, G^1)$  such that  $\text{export}(G^0) = \text{export}(G^1)$

<sup>2</sup>In the SSProve formalization, composition can actually be performed on arbitrary packages, but the obtained packages are guaranteed to be valid only when the requirements stated here are met, as detailed in §3.3.

and  $\text{import}(G^0) = \text{import}(G^1) = \emptyset$ . A *distinguisher* for a game pair is a package  $\mathcal{D}$  with  $\text{import}(\mathcal{D}) = \text{export}(G^0) = \text{export}(G^1)$  and  $\text{export}(\mathcal{D}) = \{\text{Run}\}$ , where  $\text{Run}$  is an entry-point procedure that can call the procedures exported by the games and returns a boolean value: true or false. When a game  $G$  (so without imports) exports a single procedure  $\text{Run} : \text{unit} \rightarrow \text{bool}$  as above, we denote by  $\Pr[b \leftarrow G]$  the probability that  $G.\text{Run}$  returns the boolean value  $b$  when running on initial memory. We can quantify how much a distinguisher can distinguish the two packages in a game pair:

*Definition 2.1 (Distinguisher advantage).* The advantage of a distinguisher  $\mathcal{D}$  against a game pair  $G = (G^0, G^1)$  is

$$\alpha(G)(\mathcal{D}) = |\Pr[\text{true} \leftarrow \mathcal{D} \circ G^0] - \Pr[\text{true} \leftarrow \mathcal{D} \circ G^1]|$$

*Reasoning about advantage.* Next, we review the two main results used for equational-like reasoning about advantage against games in SSP:

LEMMA 2.2 (TRIANGLE INEQUALITY). *Let  $G^0, G^1$  and  $G^2$  be games, we have that for every distinguisher  $\mathcal{D}$ ,*

$$\alpha(G^0, G^2)(\mathcal{D}) \leq \alpha(G^0, G^1)(\mathcal{D}) + \alpha(G^1, G^2)(\mathcal{D}).$$

PROOF. By unfolding Definition 2.1 we have

$$\begin{aligned} \alpha(G^0, G^2)(\mathcal{D}) &= |\Pr[\text{true} \leftarrow \mathcal{D} \circ G^0] - \Pr[\text{true} \leftarrow \mathcal{D} \circ G^2]| \\ &= |\Pr[\text{true} \leftarrow \mathcal{D} \circ G^0] - \Pr[\text{true} \leftarrow \mathcal{D} \circ G^1] \\ &\quad + \Pr[\text{true} \leftarrow \mathcal{D} \circ G^1] - \Pr[\text{true} \leftarrow \mathcal{D} \circ G^2]| \\ &\leq |\Pr[\text{true} \leftarrow \mathcal{D} \circ G^0] - \Pr[\text{true} \leftarrow \mathcal{D} \circ G^1]| \\ &\quad + |\Pr[\text{true} \leftarrow \mathcal{D} \circ G^1] - \Pr[\text{true} \leftarrow \mathcal{D} \circ G^2]| \\ &= \alpha(G^0, G^1)(\mathcal{D}) + \alpha(G^1, G^2)(\mathcal{D}) \end{aligned}$$

□

In general, we want to bound the advantage to distinguish  $G^0$  and  $G^n$  (i.e., the advantage  $\alpha(G^0, G^n)(\mathcal{D})$  against game pair  $(G^0, G^n)$ ). In order to do so, by repeatedly applying Lemma 2.2, it is enough to exhibit a chain of games  $G^0, G^1, G^2, \dots, G^n$  so that a bound for  $\alpha(G^0, G^n)(\mathcal{D})$  can be given by

$$\alpha(G^0, G^1)(\mathcal{D}) + \alpha(G^1, G^2)(\mathcal{D}) + \dots + \alpha(G^{n-1}, G^n)(\mathcal{D}).$$

LEMMA 2.3 (REDUCTION). *Let  $(G^0, G^1)$  be a game pair and let  $M$  be an arbitrary package. Then, for every distinguisher  $\mathcal{D}$ , we have*

$$\alpha(M \circ G^0, M \circ G^1)(\mathcal{D}) = \alpha(G^0, G^1)(\mathcal{D} \circ M).$$

PROOF. By unfolding Definition 2.1 and applying the associativity law of sequential composition (Equation (1)), we have

$$\begin{aligned} \alpha(M \circ G^0, M \circ G^1)(\mathcal{D}) &= |\Pr[\text{true} \leftarrow \mathcal{D} \circ (M \circ G^0)] - \Pr[\text{true} \leftarrow \mathcal{D} \circ (M \circ G^1)]| \\ &= |\Pr[\text{true} \leftarrow (\mathcal{D} \circ M) \circ G^0] - \Pr[\text{true} \leftarrow (\mathcal{D} \circ M) \circ G^1]| \\ &= \alpha(G^0, G^1)(\mathcal{D} \circ M) \end{aligned}$$

□

As its name indicates, Lemma 2.3 is used to reduce the advantage of the distinguisher over a composed game  $(M \circ G^b)$ , to the advantage over part of the game  $(M)$ , for which we know a bound. We will use both these SSP lemmas in §2.3.

One difference in SSProve with respect to the SSP papers is that up to this point we made no “state separation” assumptions. We proved instead in Coq that the algebraic laws for package composition as well as the two lemmas above hold even when the involved packages share state.

*Adversaries.* State separation is, however, still crucial for defining adversaries against game pairs. Formally, an *adversary*  $\mathcal{A}$  for a game pair is a distinguisher whose state is disjoint from the state of each game in the pair.

*Perfect game indistinguishability.* We say that the games  $G^0$  and  $G^1$  of a game pair are *perfectly indistinguishable* when  $\alpha(G^0, G^1)(\mathcal{A}) = 0$  for every adversary  $\mathcal{A}$ . Perfect indistinguishability is a form of observational equivalence and states that no adversary can learn any information about which game in the pair it is interacting with.

## 2.2 Proving perfect indistinguishability steps in a probabilistic relational program logic

We now present the main novel result brought by SSProve. The SSP laws above deal only with the high-level structure of composed packages. However we often also need to show that two concrete games are equivalent with respect to what an adversary can learn from using them, i.e. perfect indistinguishability. In SSProve we formally verify this kind of equivalence by reducing it to proving a family of semantic judgments in a probabilistic relational program logic. The logic we use is a variant of pRHL, a probabilistic relational Hoare logic introduced by Barthe et al. [16]. Judgments of this logic are of the form

$$\models \{(m_0, m_1). \phi\} c_0 \sim c_1 \{((m'_0, a_0), (m'_1, a_1)). \psi\},$$

and intuitively mean that after separately running the two code fragments  $c_0$  and  $c_1$  on the corresponding component of a pair of memories  $m_0, m_1$  satisfying a precondition  $\phi$ , the final memories  $m'_0, m'_1$  and results  $a_0, a_1$  satisfy the postcondition  $\psi$ . In this paper we write as  $p.M$  a function that binds  $p$  and has body  $M$  (usually denoted by  $\lambda p.M$  in the functional programming community). This notation will be handy for writing postconditions, which depend on final memories and on final results. We adopt the convention that the variables  $m_0$  and  $m_1$  stand for the state associated to  $c_0$  and  $c_1$  in preconditions, the initial memories, and  $m'_0, m'_1$  stand for the corresponding state in postconditions, the final memories. We will omit them from judgments when no ambiguity can arise. We now state the main theorem of SSProve:

**THEOREM 2.4.** *Let  $G = (G^0, G^1)$  be a game pair with respect to export interface  $\mathcal{E} = \text{export}(G^0) = \text{export}(G^1)$ . Moreover, assume that  $\psi$  is a stable invariant that relates the memories of  $G^0$  and  $G^1$ , and that it holds on the initial memories.*

*If for each provided procedure  $f : A \rightarrow B \in \mathcal{E}$ , we have that for all  $a \in A$ ,*

$$\models \{\psi\} G^0.f(a) \sim G^1.f(a) \{(b_0, b_1). b_0 = b_1 \wedge \psi(m'_0, m'_1)\}$$

*then we can conclude that  $\alpha(G^0, G^1)(\mathcal{A}) = 0$  for any  $\mathcal{A}$ .*

Intuitively, we ask that both procedures, run on memories satisfying  $\psi$ , yield results drawn from the same distribution and memories still satisfying  $\psi$ . We leave the precise definition of stable invariants and how this theorem is proved to §4.2, but the main idea behind this invariant is that it keeps track of a relation between the memories of  $G^0$  and  $G^1$ , and that this relation is preserved as different procedures from the interface are called during the execution. We illustrate how this theorem is used in the examples from the next two subsections.

<pre>package: PRF<sup>0</sup> mem: k : option key</pre>	<pre>package: PRF<sup>1</sup> mem: T : map [key -&gt; key]</pre>
<pre>Eval-prf(x):   if k = ⊥ then     k &lt;\$ uniform {0,1}<sup>n</sup>   return prf(k, x)</pre>	<pre>Eval-prf(x):   if T[x] = ⊥ then     T[x] &lt;\$ uniform {0,1}<sup>n</sup>   return T[x]</pre>

Fig. 4. Packages PRF<sup>0</sup> and PRF<sup>1</sup>

### 2.3 Security proof of PRF-based encryption in SSProve

We first illustrate the key ideas of SSProve on a cryptographic proof by Brzuska et al. [25] that we have verified in Coq using our framework. In this proof, reasoning about composed packages (using Lemmas 2.2 and 2.3 above) allows for a high level of abstraction that drives the proof argument. Some steps of this proof are, however, justified by perfect indistinguishability between games, which involves inspecting the procedures of the games and applying program transformations to show the equivalence. In the previous paper proof [25] these steps were only justified *informally* by code inspection. Instead, we have *formally* verified these steps too, using Theorem 2.4 and our relational program logic.

Brzuska et al. [25] show how to construct a symmetric encryption scheme out of a pseudo-random function (PRF) and use the SSP methodology to reduce security of the encryption scheme—expressed as IND-CPA—to the security of the PRF, expressed as being *indistinguishable* from a package doing random sampling.

The scheme assumes a pseudo-random function called `prf` with the following signature

$$\text{prf} : \{0, 1\}^n \times \{0, 1\}^n \rightarrow \{0, 1\}^n$$

where  $\{0, 1\}^n$  represents the set of  $n$ -bit sequences. It is possible to formalize and quantify the security of PRF as the probability for an adversary to distinguish it from a package that samples from an uniform distribution (*real vs random* paradigm [57]). Formally, given the packages PRF<sup>0</sup> and PRF<sup>1</sup> as in Figure 4, the security of PRF,  $\alpha(\text{PRF})(\mathcal{A})$  is defined using Definition 2.1 as the advantage of an adversary for the game pair  $\text{PRF} = (\text{PRF}^0, \text{PRF}^1)$ :

$$\alpha(\text{PRF})(\mathcal{A}) = |\Pr[\text{true} \leftarrow \mathcal{A} \circ \text{PRF}^0] - \Pr[\text{true} \leftarrow \mathcal{A} \circ \text{PRF}^1]|$$

The three basic algorithms constructing a symmetric encryption scheme out of `prf` are given in Figure 5. These are not packages themselves, but rather code used inside packages.

The security property proposed for this encryption scheme is defined as the advantage on a game pair that captures indistinguishability under chosen-plaintext attack (IND-CPA). We refer to this game pair as  $(\text{IND-CPA}^0, \text{IND-CPA}^1)$ , and the packages involved are introduced in Figure 6. Notice that in procedure  $\text{IND-CPA}^1.\text{Enc}$  the argument `msg` is never used, the encryption procedure is run on a random `msg'`. Therefore the advantage of an adversary with respect to the game  $(\text{IND-CPA}^0, \text{IND-CPA}^1)$  represents the probability that the adversary is able to distinguish the encryption of `msg` from the encryption of a random bit-string. The security of the encryption procedure with respect to an adversary  $\mathcal{A}$  is then  $\alpha(\text{IND-CPA})(\mathcal{A})$ .

Brzuska et al. [25] use a sequence of *game-hops* to bound  $\alpha(\text{IND-CPA})$  in terms of (a linear function of) the advantage  $\alpha(\text{PRF})$ . This technique of game-hops follows the style of inequality reasoning chains from §2.1 (Lemma 2.2), where each step involves establishing the advantage on a game pair,



```

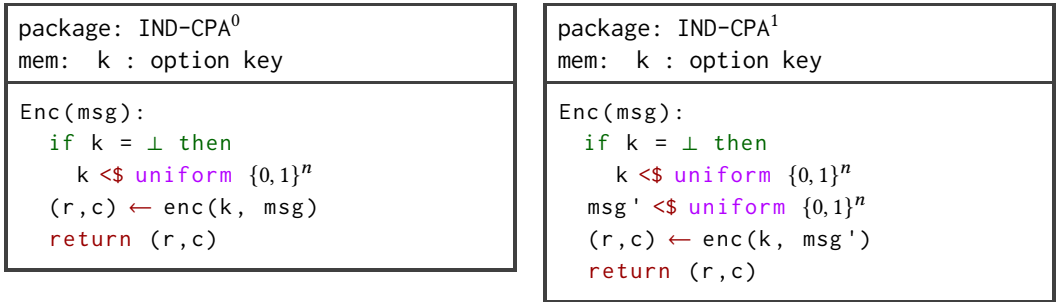
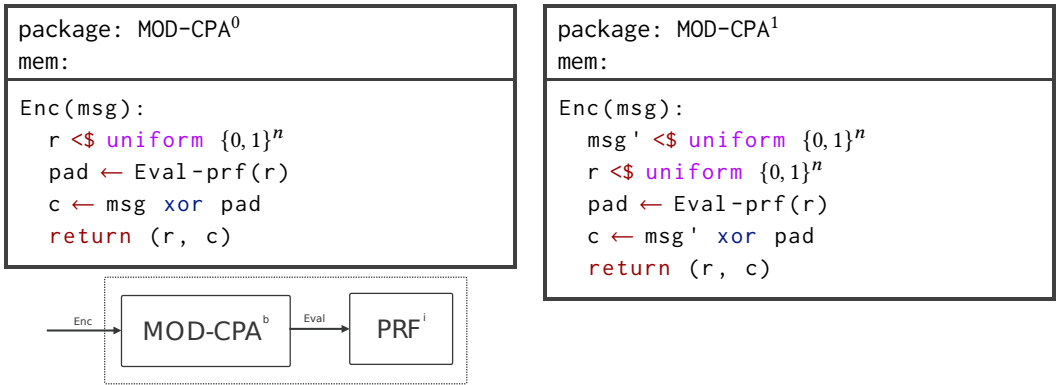
enc(k, msg):
  r <$ uniform {0,1}n
  pad ← prf(k, r)
  c ← msg xor pad
  return (r, c)

kgen():
  k <$ uniform {0,1}n
  return k

dec(k, (r, c)):
  pad ← prf(k, r)
  msg ← c xor pad
  return msg

```

Fig. 5. Algorithms for prf-based encryption scheme

Fig. 6. Packages IND-CPA<sup>0</sup> and IND-CPA<sup>1</sup>Fig. 7. Packages MOD-CPA<sup>b</sup> import Eval-prf from PRF<sup>i</sup>

and as a result we obtain a bound on the advantage of the game consisting of the initial and final game.

In this example, IND-CPA<sup>b</sup> is shown equivalent to a variant, MOD-CPA<sup>b</sup>, that gets the secret key through the PRF, i.e. with a call to Eval-prf of the package PRF<sup>0</sup> or PRF<sup>1</sup> (see Figure 7). By repeatedly

applying Lemma 2.2, we bound  $\alpha(\text{IND-CPA})(\mathcal{A})$  by

$$\begin{aligned} & \alpha(\text{IND-CPA}^0, \text{MOD-CPA}^0 \circ \text{PRF}^0)(\mathcal{A}) + \\ & \alpha(\text{MOD-CPA}^0 \circ \text{PRF}^0, \text{MOD-CPA}^0 \circ \text{PRF}^1)(\mathcal{A}) + \\ & \alpha(\text{MOD-CPA}^0 \circ \text{PRF}^1, \text{MOD-CPA}^1 \circ \text{PRF}^1)(\mathcal{A}) + \\ & \alpha(\text{MOD-CPA}^1 \circ \text{PRF}^1, \text{MOD-CPA}^1 \circ \text{PRF}^0)(\mathcal{A}) + \\ & \alpha(\text{MOD-CPA}^1 \circ \text{PRF}^0, \text{IND-CPA}^1)(\mathcal{A}) \end{aligned}$$

By observing that  $\alpha(\text{IND-CPA}^0, \text{MOD-CPA}^0 \circ \text{PRF}^0)(\mathcal{A}) = 0$ , and  $\alpha(\text{MOD-CPA}^1 \circ \text{PRF}^0, \text{IND-CPA}^1)(\mathcal{A}) = 0$ , and by using Lemma 2.3 twice, we reduce this bound to

$$\alpha(\text{PRF})(\mathcal{A} \circ \text{MOD-CPA}^0) + \varepsilon_{\text{stat.}}(\mathcal{A}) + \alpha(\text{PRF})(\mathcal{A} \circ \text{MOD-CPA}^1).$$

where  $\varepsilon_{\text{stat.}} = \alpha(\text{MOD-CPA}^0 \circ \text{PRF}^1, \text{MOD-CPA}^1 \circ \text{PRF}^1)$ . The advantage of an attacker with respect to  $\text{MOD-CPA}^0$  and  $\text{MOD-CPA}^1$  is usually referred to as *statistical gap*, a polynomial function of the number of calls from the adversary (see [25, appendix A]).

It remains to justify the two perfect indistinguishabilities stated above. These steps involve replacing an informal argument [25] by a fully formal one, moving to our probabilistic relational program logic. We will detail one of these steps:  $\alpha(\text{IND-CPA}^0, \text{MOD-CPA}^0 \circ \text{PRF}^0)(\mathcal{A}) = 0$ . The other step,  $\alpha(\text{MOD-CPA}^1 \circ \text{PRF}^0, \text{IND-CPA}^1)(\mathcal{A}) = 0$ , is analogous.

In order to prove this equivalence, Brzuska et al. [25] notice that the Enc procedures of  $\text{IND-CPA}^0$  and  $\text{MOD-CPA}^0 \circ \text{PRF}^0$  (see Figure 8) return the same ciphertext when called on the same msg. The two procedures are obtained by “inlining” the code of  $\text{PRF}^0.\text{Eval-prf}$  inside  $\text{MOD-CPA}^0$ , and by “unfolding” the code of enc.

IND-CPA <sup>0</sup> .Enc(msg)	(MOD-CPA <sup>0</sup> ◦ PRF <sup>0</sup> ).Enc(msg)
<pre style="margin: 0;">if k = ⊥ then   k &lt;\$ uniform {0,1}<sup>n</sup> r &lt;\$ uniform {0,1}<sup>n</sup> pad ← prf(k, r) c ← msg xor pad return (r, c)</pre>	<pre style="margin: 0;">r &lt;\$ uniform {0,1}<sup>n</sup> if k = ⊥ then   k &lt;\$ uniform {0,1}<sup>n</sup> pad ← prf(k, r) c ← msg xor pad return (r, c)</pre>

Fig. 8. ENC procedures expanded

The left- and right-hand side procedures in Figure 8 only differ when  $k = \perp$ , in which case the left Enc procedure first samples  $k$  and then  $r$ , while the right Enc first samples  $r$  and then  $k$ . In both procedures,  $k$  and  $r$  are drawn from *independent* distributions. Here Brzuska et al. [25] conclude informally that independence allows to “swap” the two operations. We instead use Theorem 2.4 to formally reduce  $\alpha(\text{IND-CPA}^0, \text{MOD-CPA}^0 \circ \text{PRF}^0)(\mathcal{A}) = 0$  to showing the equivalence of the two Enc procedures from Figure 8. In our probabilistic relational program logic, this comes down to proving the following judgment for all plaintext messages msg,

$$\begin{aligned} & \models \{m_0 = m_1\} \\ & \text{IND-CPA}^0.\text{Enc}(\text{msg}) \sim (\text{MOD-CPA}^0 \circ \text{PRF}^0).\text{Enc}(\text{msg}) \\ & \{(c_0, c_1). m'_0 = m'_1 \wedge c_0 = c_1\}. \end{aligned}$$

This judgment intuitively states that encrypting msg with the same initial memories “ $m_0 = m_1$ ”, terminates in memories and ciphertexts drawn from the same distribution, “ $m'_0 = m'_1 \wedge c_0 = c_1$ ”. We use the following instance of the swap rule from §4.1, to formally justify this swapping:

```

KeyGen():
  sk <$ uniform {0, ..., n-1}
  pk ← gsk
  return (pk, sk)

Enc(pk, msg):
  rnd <$ uniform {0, ..., n-1}
  crnd ← grnd
  cmsg ← msg * pkrnd
  return (crnd, cmsg)

Dec(sk, (crnd, cmsg)):
  return cmsg * (crndsk)-1

```

Fig. 9. Algorithms for ElGamal encryption scheme

$$\begin{array}{c}
\vdash \{m_0 = m_1\} \ k \leftarrow \text{uniform}\{0,1\}^n \sim r \leftarrow \text{uniform}\{0,1\}^n \ \{m'_0 = m'_1 \wedge c_0 = c_1\} \\
\vdash \{m_0 = m_1\} \ r \leftarrow \text{uniform}\{0,1\}^n \sim k \leftarrow \text{uniform}\{0,1\}^n \ \{m'_0 = m'_1 \wedge c_0 = c_1\} \\
\hline
\vdash \{m_0 = m_1\} \\
k \leftarrow \text{uniform}\{0,1\}^n ; r \leftarrow \text{uniform}\{0,1\}^n \\
r \leftarrow \text{uniform}\{0,1\}^n ; k \leftarrow \text{uniform}\{0,1\}^n \\
\{m'_0 = m'_1 \wedge c_0 = c_1\}
\end{array}$$

## 2.4 Security proof of ElGamal in SSProve

We also illustrate the key ideas of SSProve on a security proof for the ElGamal encryption scheme inspired by *The Joy of Cryptography* textbook [57, Chapter 15.3]. ElGamal belongs to the family of *public-key* or *asymmetric* encryption schemes, which use a public key for encryption and a private one for decryption. Public-key schemes therefore require a key generation algorithm producing the pair of public and private keys. In our formalization it suffices to provide the aforementioned algorithms together with key-, plaintext- and cipher-spaces to automatically obtain a public-key scheme together with its related security notions (to be proved) such as security against chosen plaintext attacks (CPA). In what follows we describe which spaces and algorithms define ElGamal and the security proof we provided for it.

ElGamal is parameterized by a multiplicative cyclic group  $(\mathcal{G}, *)$  with  $n$  elements and with generator  $g$ , usually denoted by  $\langle g \rangle = \mathcal{G}$ . Plaintexts are elements  $msg \in \mathcal{G}$  and ciphertexts are pairs of group elements  $c = (c_{rnd}, c_{msg}) \in \mathcal{G} \times \mathcal{G}$ . Secret keys are elements of  $\mathbb{Z}_n$ , while public keys are group elements once again,  $pk \in \mathcal{G}$ . The key generation algorithm (KeyGen in Figure 9) generates a secret key that is a random number  $sk \in \{0, \dots, n-1\}$  and a public key that is  $g^{sk}$ . Encryption and decryption (Enc and Dec in Figure 9) involve the group operation  $(\_*)$ , exponentiation  $(\_)$ - and the multiplicative inverse  $(\_)^{-1}$ .

Under the *Decisional Diffie-Hellman* (DDH) assumption for the group  $\mathcal{G}$ , namely that  $DDH^0$  and  $DDH^1$  from Figure 10 are computationally indistinguishable, one can prove that an adversary cannot distinguish messages encrypted with the ElGamal scheme from ciphertexts that are randomly sampled (CPA). Our formalization only considers the case in which the adversary can see a single ciphertext (*one-time* CPA, written OT-CPA), as it is known that this suffices for public-key encryption schemes to satisfy CPA [57, Claim 15.5]. We leave the formalization of this last result as future work and discuss hereafter our proof of OT-CPA in SSProve.

<pre>package: DDH<sup>0</sup> mem: pk : option pubKey       sk : option secKey</pre>	<pre>package: DDH<sup>1</sup> mem: pk : option pubKey       sk : option secKey</pre>
<pre>Query()   sk &lt;\$ uniform {0, ..., n-1}   rnd &lt;\$ uniform {0, ..., n-1}   pk ← g<sup>sk</sup>   return (g<sup>sk</sup>, g<sup>rnd</sup>, g<sup>sk·rnd</sup>)</pre>	<pre>Query()   sk &lt;\$ uniform {0, ..., n-1}   rnd &lt;\$ uniform {0, ..., n-1}   rnd' &lt;\$ uniform {0, ..., n-1}   pk ← g<sup>sk</sup>   return (g<sup>sk</sup>, g<sup>rnd</sup>, g<sup>rnd'</sup>)</pre>

Fig. 10. The DDH assumption states that  $DDH^0$  and  $DDH^1$  are computationally indistinguishable

The security property OT-CPA is expressed in terms of the advantage against game pair  $(CPA^0, CPA^1)$  in Figure 11. An adversary  $\mathcal{A}$  can call `Get_pk()` and get the public key,<sup>3</sup> if already initialized. The adversary can “challenge” a package to encrypt a certain plaintext `msg` through `Challenge(msg)`. Both packages return a ciphertext only if the counter is 0—as expressed by the use of `assert`—so the adversary can only see one ciphertext. Both packages call `KeyGen` to generate public and private keys, but while  $CPA^0$  indeed encrypts the message provided by the adversary with the public key through `Enc(pk, msg)`, the package  $CPA^1$  instead returns a randomly sampled ciphertext  $(c_{rnd}, c_{msg}) \leftarrow \text{uniform } \mathcal{G} \times \mathcal{G}$ , i.e. a pair of group elements sampled from the uniform distribution on  $\mathcal{G} \times \mathcal{G}$ .

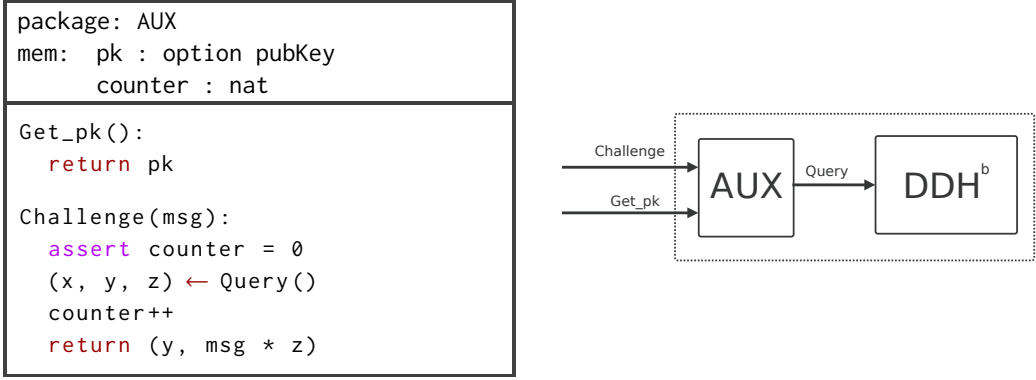
<pre>package: CPA<sup>0</sup> mem: pk : option pubKey       sk : option secKey       counter : nat</pre>	<pre>package: CPA<sup>1</sup> mem: pk : option pubKey       sk : option secKey       counter : nat</pre>
<pre>Get_pk():   return pk</pre> <pre>Challenge(msg):   assert counter = 0   (pk, sk) ← KeyGen()   (c<sub>rnd</sub>, c<sub>msg</sub>) ← Enc(pk, msg)   counter++   return (c<sub>rnd</sub>, c<sub>msg</sub>)</pre>	<pre>Get_pk():   return pk</pre> <pre>Challenge(msg):   assert counter = 0   (pk, sk) ← KeyGen()   (c<sub>rnd</sub>, c<sub>msg</sub>) &lt;\$ uniform <math>\mathcal{G} \times \mathcal{G}</math>   counter++   return (c<sub>rnd</sub>, c<sub>msg</sub>)</pre>

Fig. 11. Packages  $CPA^0$  and  $CPA^1$  in ElGamal

The OT-CPA proof reduces the advantage of adversary  $\mathcal{A}$  against  $(CPA^0, CPA^1)$  to the advantage of  $\mathcal{A} \circ \text{AUX}$  against  $(DDH^0, DDH^1)$ , with the auxiliary package `AUX` listed in Figure 12:

$$\alpha(CPA)(\mathcal{A}) \leq \alpha(DDH)(\mathcal{A} \circ \text{AUX}).$$

<sup>3</sup>In a previous version of this work we were – erroneously – not providing `Get_pk()`, the result was not a proper public-key scheme.

Fig. 12. Package AUX imports Query from  $DDH^b$ 

We once again obtain this result by repeatedly applying Lemma 2.2 to bound  $\alpha(CPA)(\mathcal{A})$  by

$$\begin{aligned} & \alpha(CPA^0, AUX \circ DDH^0)(\mathcal{A}) + \\ & \alpha(AUX \circ DDH^0, AUX \circ DDH^1)(\mathcal{A}) + \\ & \alpha(AUX \circ DDH^1, CPA^1)(\mathcal{A}) \end{aligned}$$

We will see that the first and last advantages are null by proving the packages perfectly indistinguishable, and the remaining advantage is equal to  $\alpha(DDH)(\mathcal{A} \circ AUX)$  by simple application of Lemma 2.3. It now remains to show the equivalences below:

*Step  $\alpha(CPA^0, AUX \circ DDH^0)(\mathcal{A}) = 0$ :* We apply Theorem 2.4 and reduce the goal to a relational judgment between  $CPA^0.Challenge(msg)$  and  $(AUX \circ DDH^0).Challenge(msg)$  for a generic plaintext  $msg$ , and where the invariant  $\psi$  is equality of memories. Inlining the code of Query provided by  $DDH^0$  inside AUX and unfolding one realizes that the two code fragments are identical and the judgment holds by application of the reflexivity rule in §4.1.

*Step  $\alpha(AUX \circ DDH^1, CPA^1)(\mathcal{A}) = 0$ :* This step is quite similar to the one above. After inlining however the two code fragments are not exactly the same, since in particular  $CPA^1$  completely ignores  $msg$  and returns a random ciphertext, while  $AUX \circ DDH^1$  returns  $msg * g^{rnd'}$  for a random  $rnd'$ . To have equality of memories as invariant  $\psi$ , we show that in  $\mathcal{G}$ , multiplication by  $g^{\cdot}$  acts like a one time pad, which is a standard result [16, Section 6.2].

### 3 FORMALIZING STATE-SEPARATING PROOFS

We separate the programming language and thus the reasoning into two strata: code and packages. We define the syntax of code (§3.1), relate it to the notation used in §2.1, and explain its semantics (§3.2). We then give a formal description of packages (§3.3) and the algebraic laws they obey (§3.4). We took some license regarding notation in §2.1 in order to stay close to the presentation of [25]. The code examples in the remainder of the paper faithfully follow the Coq notations we use in the formal development of SSProve.

#### 3.1 Syntax for cryptographic code (free monad)

The language of the Coq system, Gallina, is a dependently typed, purely functional programming language. As such, we can directly express functional code in Gallina, but not code with side-effects such as reading from and writing to memory, probabilistic sampling, or external procedure calls.

We thus represent cryptographic code via a combination of the ambient language Gallina and a monad of effectful computations. Monads constitute an established way of adding effects to a purely functional language [49, 64]. Free monads in particular allow to separate the representation (syntax) of an embedded language from its interpretation (semantics).

*Raw code.* We use a hybrid approach [51] of embedding the pure fragment of our cryptographic programming language shallowly in Coq, and embedding the effects deeply via a free monad. This free monad is defined as an inductive type:

```
Inductive raw_code A : Type :=
| return (x : A)
| call (p : opsig) (x : src p) (κ : tgt p → raw_code A)
| get (ℓ : Location) (κ : type ℓ → raw_code A)
| put (ℓ : Location) (v : type ℓ) (κ : raw_code A)
| sample (op : Op) (κ : Arit op → raw_code A).
```

This type of raw code comes equipped with an induction principle, which is used for instance in the proof of Theorem 2.4, in Theorem 4.1, and in the definition of the bind operation and sequential composition of packages by recursion over code.

Some more explanations about `raw_code` are in order. The type parameter `A` indicates the result of a computation. The first clause of the above definition lets us inject any pure value `x` of type `A` into the monad as `return x`. Calls to external procedures are represented via `call p x κ`, where `p : opsig` specifies the name of the procedure, the type of its argument (`src p`), and its return type (`tgt p`). The last argument `κ` is the continuation of the program, awaiting the result of the call to `p`. The `get` and `put` operations take a (typed) location `ℓ` as argument, respectively read from and write to that location, and continue with the continuation `κ`. Finally, we may sample from a collection of probabilistic subdistributions `Op`. Subdistributions constitute the base of our code semantics and are further discussed in §3.2. The type `Op` is a parameter of the language that can be instantiated by the user. Sampling a subdistribution `op` on type `Arit op` can be composed with a matching continuation `κ` (continuations are explained below).

We will use the following two pieces of code as running examples to explain different aspects of the definition.

$$\text{get } \ell (\lambda x_\ell . \text{put } \ell (x_\ell + 1) (\text{return } x_\ell)) \quad (2)$$

$$\text{sample } (\text{uniform } \{0, 1\}^n) \\ (\lambda y . \text{call Prf } (y, 101010) (\lambda z . \text{return } z)) \quad (3)$$

The code in (2) increments the value stored at location `ℓ` by 1 and returns the value before the increment. The code in (3) draws a random bit-string `y` of length `n`, calls an external procedure `Prf` with arguments `y` and `101010`, and returns the result.

*Valid code.* Raw code is merely a representation of syntax. To record which probabilistic sampling operations, imported procedures, and locations are used, we introduce a notion of valid code. Validity is defined relatively to a collection of sampling operations `Op`, a set of locations `ℒ`, and finally an import *interface* `I` which is a set of procedure signatures (`opsig`) consisting of a name, an input type and an output type. Concretely, the code in (2) is valid with respect to  $\{\ell : \text{nat}\}$  and the empty import interface, while (3) is valid with respect to the empty set of locations and the interface  $\{\text{Prf} : \text{nat} \times \text{nat} \rightarrow \text{nat}\}$ , assuming further that `uniform {0, 1}n : Op` is a valid sampling operation. The type code is then simply defined as valid raw code:

$$\text{code}_{\mathcal{L}, I} A = \{c : \text{raw\_code } A \mid \text{is\_valid } c \ \mathcal{L} \ I\}$$

In the paper we sometimes omit the set of locations and the interface. Thanks to the use of tactics and Coq’s type classes, proofs of validity for well-scoped user-written code are constructed automatically without requiring user intervention.

*Continuations.* A *continuation* is a suspended computation awaiting the result of an operation, intuitively corresponding to the rest of the program. Consider for instance the code (2). The `get` operation performs a memory lookup at the location  $\ell$ , and its continuation is a Coq function  $(\lambda x_\ell . \text{put } \dots)$  of type  $(\text{type } \ell \rightarrow \text{raw\_code nat})$  which receives the value stored at  $\ell$  as its parameter  $x_\ell$ . The continuation in turn performs a `put` operation, storing the value  $x_\ell + 1$  at memory location  $\ell$ , and returns the value  $x_\ell$ . The code thus corresponds to the expression commonly written as  $\ell++$ .

*Variables.* As demonstrated in example (2), we draw a strict distinction between a location  $\ell$ , which can be accessed and updated via `get` and `put`, and the value stored in memory at location  $\ell$ . In (2), this value is available in the continuation of `get`  $\ell (\lambda x_\ell . \text{put } \dots)$  as  $x_\ell$ . Formally speaking,  $x_\ell$  is an immutable Coq variable, and in (2) the location  $\ell$  itself is a Coq variable of type `Location`. This distinction is already present in [25, Def. 2], where locations correspond to “state variables” and the ambient, mathematical notion of variable is referred to as “local variable”.

*(Un-)initialised memory cells.* We frequently have to consider the situation where a memory location  $\ell$  is not yet initialised to a value when we first run a program, and a value is stored only after some other procedure is run, for instance to generate a key. This situation arises, for instance, in Figure 6. If we intuitively want  $\ell$  to hold a value of some type `key`, we express the fact that  $\ell$  may not yet be initialised by defining  $\ell$  such that `type`  $\ell = \text{option key}$ , using the following definition:

```
Inductive option A :=  $\perp$  | Some (a : A).
```

Values of the type `option key` can be constructed either by providing a dummy value  $\perp$  or a value of type `key`, which is tagged with the constructor `Some`. This tag allows us to have a clean distinction on whether or not a location has been initialised, instead of, say, always storing some default key value or prescribing the behaviour of the memory allocator.

*Monadic bind.* The `bind` operation of the monad, with type `code A  $\rightarrow$  (A  $\rightarrow$  code B)  $\rightarrow$  code B`, allows the composition of effectful code. Take for instance the following pieces of code.

```
Definition c : code nat :=
```

```
  sample (uniform bool) ( $\lambda b . \text{if } b \text{ then return } m_1 \text{ else return } m_2)$ )
```

```
Definition  $\kappa$  : nat  $\rightarrow$  code nat :=  $\lambda m . \text{put } \ell m (\text{return } 0)$ 
```

We would like to use `c` as an argument to  `$\kappa$` , but the types don’t match:  `$\kappa$`  expects a value of type `nat` as argument, not a computation of type `code nat`. We define a standard `bind` operation that achieves this by traversing the code of `c`, applying  `$\kappa$`  when a returned value is encountered, and recursively pushing  `$\kappa$`  into any other continuations.

```
Fixpoint bind (c : code A) ( $\kappa$  : A  $\rightarrow$  code B) : code B :=
```

```
  match c with
```

```
  | return a  $\Rightarrow$   $\kappa$  a
```

```
  | call p v  $\kappa'$   $\Rightarrow$  call p v ( $\lambda x . \text{bind } (\kappa' x) \kappa$ )
```

```
  | get l  $\kappa'$   $\Rightarrow$  get l ( $\lambda v . \text{bind } (\kappa' v) \kappa$ )
```

```
  | put l v  $\kappa'$   $\Rightarrow$  put l v (bind  $\kappa' \kappa$ )
```

```
  | sample op  $\kappa'$   $\Rightarrow$  sample op ( $\lambda a . \text{bind } (\kappa' a) \kappa$ )
```

```
end
```

An easy structural induction over code allows us to prove that `bind` satisfies the expected monad laws: `bind m (λ p . bind (f p) g) = bind (bind m f) g`, and `return` serves as a unit (`bind (return x) f = f x` and `bind m return = m`).

*Loops.* We do not have syntax for loops in code. However, since we are embedding in Coq we take advantage of its recursion mechanisms to write terminating loops. The most basic construction we can write is a “`for i := 0 to N do c`” loop that repeats (n+1)-times a command `c`, providing to `c` the value of the index `i`:

```
Fixpoint for_loop (N : nat) (c : nat → code unit): code unit :=
  match N with
  | 0 ⇒ c 0
  | S i ⇒ bind (for_loop i c) (λ _ . c N)
  end.
```

More generally, we can define a “do-while” loop that repeatedly executes a loop body while a condition holds, checked after each iteration. To ensure termination in Coq we add a natural number `N` to bound the maximum number of iterations:

```
Fixpoint do_while (N : nat) (c : code bool) : code bool :=
  match N with
  | 0 ⇒ return false
  | S n ⇒ bind c (λ b. if b then do_while n c else return true)
  end.
```

At the end, the returned boolean signals whether there was remaining fuel (i.e. iteration steps) available or not.

*Standard subdistributions.* Probabilistic operations denoting a collection of subdistributions we may sample from are included in the parameter type `Op`. Standard subdistributions including uniform sampling on finite types as well as a null subdistribution are predefined for convenience. The null subdistribution in particular allows us to represent failure and an `assert` construct. Failure at type `A` is defined as sampling from the null distribution `dnull`.

```
Definition fail A : code A := x ← sample (A ; dnull) ; return x.
```

A simple assertion is not expected to produce any interesting values but only gets evaluated for the possibility of failing if the condition is violated. This is expressed by the fact that a successful `assert` simply returns a value of unit type, where unit stands for the Coq singleton type with a unique inhabitant `()`.

```
Definition assert (b : bool) : code unit := if b then return () else fail unit.
```

If `b` is `true`, then `assert` returns the trivial value `()`, but if `b` is `false`, we instead sample from the `dnull` subdistribution via `fail unit`, assigning probability zero to all values of the type `unit` (i.e. to `()`). Sampling from the null subdistribution is similar to non-termination, and means that the continuation will never be called.

We can see this use of `assert` in Figure 11. The packages `CPAb` ensure that the Challenge procedure can be called only once by calling `assert counter = 0`. If the assertion succeeds, we may assume that `counter = 0` holds in the rest of the procedure, until `counter` is incremented. This mode of use of `assert` is “logical”: we limit the input to certain functions or the ways in which protocols can be called in order to exploit these assertions in our security reasoning.

A different use of assertions which we may call “dependent” occurs in the following example. Consider the situation where we have a memory location `ℓ : Location` holding a key which is



not initialised. Formally, this amounts to the information type  $\ell = \text{option key}$ , which may be presented in the signature of a package as  $\text{mem}: \ell : \text{option key}$ . We did not distinguish immutable variables and locations in §2, but we carry out a careful analysis of uninitialised memory in the KEM-DEM case study in §6. For instance, in Figure 14 we implement a `Get()` procedure that returns the key stored at location `k_loc`. This procedure defines a partial function of type  $\text{unit} \rightarrow \text{key}$ , that may fail to return a value if the memory location has not yet been initialised.

```

mem: k_loc : option key
-----
Get():
  k ← get k_loc
  assert (k ≠ ⊥) as kSome
  return (getSome k kSome)

```

We first retrieve the potentially uninitialised key from memory as `k`. At this point, `k` still has type `option key`. We then check via dependent `assert` that `k` is not the uninitialised dummy value  $\perp$ , and record the asserted condition as `kSome : k ≠ ⊥`. We can now apply `getSome` with `kSome` to safely coerce `k` from `option key` to `key`. In this example, the `code` that follows the assertion depends on the asserted condition, whereas in the previous example with `counter` the assertion was only used in the logical reasoning. Indeed the continuation of dependent `assert`, called `k` in the definition below, has type  $b = \text{true} \rightarrow \text{code } A$ . Using familiar “set-builder” notation, the type of `k` can be written as  $\{\text{code } A \mid b\}$ . In other words, it constitutes a piece of code, computing a value of type `A`, that is defined only when the assertion `b` is true.

**Definition** `assertD`  $A$   $b$  ( $k : b = \text{true} \rightarrow \text{code } A$ ) : `code`  $A$  :=  
`(if b as b' return b = b' → code A then k else λ _, fail) erefl.`

This definition captures the intuition of `Get()` as a partial function, defined only under the assumption that `k_loc` has been initialised.

*Procedure calls.* A call to an external procedure such as `Prf` in (3) is represented by the `call` operation, taking as arguments a procedure name  $p$  annotated with a type, a value matching the argument type of  $p$ , and a continuation  $\kappa$  matching the return type of  $p$ . In §3.3 we show how an implementation gets substituted for this placeholder via sequential packages composition.

*Notation.* The use of continuations is pervasive in monadic code, and to alleviate the presentation we introduce the following more familiar notation. In code listings we will frequently omit the `;` separator when it would occur at the end of a line.

<code>return v</code>	<code>:=</code>	<code>return v</code>
<code>x ← c<sub>1</sub> ; c<sub>2</sub></code>	<code>:=</code>	<code>bind c<sub>1</sub> (λx. c<sub>2</sub>)</code>
<code>x ← p(a) ; c</code>	<code>:=</code>	<code>call p a (λx. c)</code>
<code>x ← get ℓ ; c</code>	<code>:=</code>	<code>get ℓ (λx. c)</code>
<code>put ℓ := v ; c</code>	<code>:=</code>	<code>put ℓ v c</code>
<code>x &lt;\$ D ; c</code>	<code>:=</code>	<code>sample D (λx. c)</code>
<code>assert e as H ; c</code>	<code>:=</code>	<code>assertD e (λH. c)</code>

*Type safety.* The typing constraints imposed by the `raw_code` definition enforce type-safety for user-written code, guaranteeing that operations and their continuations are compatible. For instance, let the continuation of `get` in (2) be  $f$ . Then  $f$  is only compatible with  $\ell$  if its domain matches the type of  $\ell$ , i.e.  $f : \text{type } \ell \rightarrow \text{raw\_code } A$  for some type  $A$ .

To see the full definition in action, we restate the procedure  $\text{Eval}(x)$  from Figure 4 more formally.

```

Definition  $\text{Eval}(x : \{0,1\}^n) : \text{raw\_code } \{0,1\}^n :=$ 
  val_k_opt  $\leftarrow$  get k ;
  val_k  $\leftarrow$  (match val_k_opt with
    |  $\perp \Rightarrow$  y  $\leftarrow$  uniform  $\{0,1\}^n$  ;
      put k := Some y ;
      return y
    | Some val_k'  $\Rightarrow$  return val_k'
  end) ;
  val_prf  $\leftarrow$  Prf(val_k, x) ;
  return val_prf.

```

Here we freely mix constructors of `raw_code` with other Gallina terms such as the `match` construct. The result of the `match` is made available to the continuation of the code as `val_k` via a use of `bind` (under the guise of '`val_k  $\leftarrow$  ... ; ...`').

### 3.2 Semantics of cryptographic code

When no external procedure calls (`call o x k`) appear in a piece of code  $c : \text{code } A$ , it is possible to interpret  $c$  as a state-transforming probability subdistribution of type

$$\text{Pr\_code } c : \text{mem} \rightarrow \text{SD}(A \times \text{mem})$$

This semantics is similar to that of Barthe et al. [17]. The type  $\text{SD } A$  denotes the collection of all subdistributions over type  $A$ . Generally speaking, a subdistribution is a function  $d : A \rightarrow \mathbb{R}$  assigning a certain probability  $d(a)$  to each  $a : A$  in such a way that  $\int_A d \leq 1$ . We use the definition of subdistributions from `mathcomp-analysis` [3, 44], a Coq library for real analysis. The semantics function `Pr_code` is defined by recursion on the structure of  $c$ . Its definition basically boils down to providing an effect handler that interprets state and probabilities in the monad  $\text{mem} \rightarrow \text{SD}(- \times \text{mem})$ .

Using this subdistribution semantics, we can formalize the notation  $\text{Pr}[b \leftarrow G]$  from §2.1 as follows: (i) extract the `Run` function from  $G$ ; (ii) apply `Pr_code` to it; (iii) run it on the initial memory; (iv) extract the boolean component (first projection) from the resulting subdistribution. The final result has type  $d : \text{SD } \text{bool}$ , the type of subdistributions for booleans, and we precisely define  $\text{Pr}[b \leftarrow G] = d(b)$  as the probability assigned to  $b$  by this subdistribution on booleans.

### 3.3 Packages

A raw package is a finite map from names to raw procedures. An *interface* is a finite set of operation signatures (`opsig`), each specifying the name, argument type, and result type of a procedure. A *package* is then a raw package  $RP$  together with an import interface  $\mathcal{I}$ , an export interface  $\mathcal{E}$ , and a set of locations  $\mathcal{L}$ , such that each procedure in  $RP$  is valid with respect to  $\mathcal{L}$  and  $\mathcal{I}$ , and each procedure name listed in  $\mathcal{E}$  is implemented by a procedure in  $RP$  of the appropriate type. Consider for instance the package `AUX` in Figure 13. The memory used,  $\text{mem}(\text{AUX})$ , consists of two locations, `pk : option publicKey` and `counter : nat`. The import interface `import(AUX)` contains a single procedure `Query : unit  $\rightarrow$   $\mathcal{G} \times \mathcal{G} \times \mathcal{G}$` . There are two procedures implemented by `AUX`, yielding an export interface `export(AUX)` containing `Get_pk : unit  $\rightarrow$  option publicKey` and `Challenge :  $\mathcal{G} \rightarrow$  option ( $\mathcal{G} \times \mathcal{G}$ )`.

We define composition of packages, following Brzuska et al. [25]. Given two raw packages  $P, Q$  we may define their *sequential composition*  $Q \circ P$  by traversing  $Q$  and replacing each `call` by the corresponding procedure implementation in  $P$ . In case  $P$  does not implement the searched for procedure, we use a dummy value instead. If the exports of  $P$  match the imports of  $Q$ , i.e.

```

package: AUX
mem: pk : option pubKey
     counter : nat

Get_pk():
  return pk

Challenge(msg):
  assert counter = 0
  (x, y, z) ← Query()
  counter++
  return (y, msg * z)

```

Fig. 13. Package AUX (repeated)

$\text{import}(Q) \subseteq \text{export}(P)$ , and both packages are valid, then so is  $Q \circ P$ , in which case no dummy value is needed. Concretely, during the traversal each call  $\text{p a } \kappa$  node is replaced by

$$\text{bind } (P. \text{p a}) (\lambda x. \text{link}_P(\kappa x))$$

where  $\text{link}_P$  stands for the recursive call of the function composing  $P$  with the remaining code. Experts will recognize this transformation as an algebraic effect handler, interpreting the free monad for probabilities, state and the operations imported by  $P$  to code in the free monad for probabilities, state, and the operations imported by  $Q$ . We have  $\text{mem}(Q \circ P) = \text{mem}(P) \cup \text{mem}(Q)$ ,  $\text{import}(Q \circ P) = \text{import}(P)$  and  $\text{export}(Q \circ P) = \text{export}(Q)$ .

Given two raw packages  $P$  and  $Q$  we may define their *parallel composition*  $P \parallel Q$  by aggregating the implementations and delegating calls to the respective package providing it. This operation is defined even if both packages have overlapping export signatures, in which case procedures in  $P$  will be given priority. If they are both valid and their exports are disjoint, i.e.  $\text{export}(P) \cap \text{export}(Q) = \emptyset$ , then this overlap situation does not happen and  $P \parallel Q$  is also valid. We have  $\text{mem}(P \parallel Q) = \text{mem}(P) \cup \text{mem}(Q)$ ,  $\text{import}(P \parallel Q) = \text{import}(Q) \cup \text{import}(P)$  and  $\text{export}(P \parallel Q) = \text{export}(Q) \cup \text{export}(P)$ .

*Private state.* When formalizing composition in SSProve we do not impose restrictions on the disjointness of the state that  $P$  and  $Q$  manipulate. The two lemmas from §2.1 and the SSP package laws below hold without any such assumptions. The essence of state separation can be thus viewed as disjointness of state between the adversary and the games in a pair. We therefore introduce the more economical assumption that *only the adversary* has to have disjoint state in our security definitions (e.g., perfect indistinguishability from §2.1) and corresponding theorem statements (e.g., Theorem 2.4).

Thanks to this finer-grained state separation, we not only remove some of the burden of formally proving disjointness, but we are also able to prove more meaningful final results. For instance, in the PRF example, enforcing state separation for all intermediary packages would mean in particular requiring the adversary to have disjoint state from  $\text{PRF}^1$ , which is just an intermediary game used within our proof. In SSProve such proof internals don't leak into the final security statements.

### 3.4 Package laws

We formally proved the algebraic laws obeyed by packages as stipulated by Brzuska et al. [25]. Sequential composition is associative and parallel composition is commutative and associative, so

for any packages  $P_1, P_2, P_3$ :

$$\begin{aligned} P_1 \circ (P_2 \circ P_3) &= (P_1 \circ P_2) \circ P_3 \\ P_1 \parallel P_2 &= P_2 \parallel P_1 \\ P_1 \parallel (P_2 \parallel P_3) &= (P_1 \parallel P_2) \parallel P_3. \end{aligned}$$

We furthermore relate the two package operations with an interchange law stating

$$(P_1 \circ P_3) \parallel (P_2 \circ P_4) = (P_1 \parallel P_2) \circ (P_3 \parallel P_4).$$

Commutativity of parallel composition only holds if the packages have indeed disjoint interfaces:  $\text{export}(P_1) \cap \text{export}(P_2) = \emptyset$ . The interchange law will only ask this of  $P_3$  and  $P_4$ :  $\text{export}(P_3) \cap \text{export}(P_4) = \emptyset$ .

The identity package  $\text{ID}_I$  behaves as an identity for sequential composition when using the correct interface:

$$\text{ID}_{\text{export}(P)} \circ P = P = P \circ \text{ID}_{\text{import}(P)}.$$

As we have hinted before, these laws do not require disjointness of state, because they are syntactic equalities. In fact, in SSProve they hold with respect to the usual equality of Coq (“propositional equality”, written  $_ = _$ ), without the need to define a separate notion of “code equality” [25].

## 4 PROBABILISTIC RELATIONAL PROGRAM LOGIC

Some of the SSP proof steps can be carried out at a high-level of abstraction relying on the package formalism from §3. The justification of other steps like perfect indistinguishability requires, however, a finer, lower-level analysis. As already pointed out in §2.2, we can perform such analyses in a relational program logic, a deductive system in which it is possible to show that two pieces of code  $c_0, c_1$  satisfy a certain relational specification, e.g. that they are equivalent.

In §4.1 we present some of the elementary rules constituting our program logic. We then sketch a proof of Theorem 2.4, the link between the high-level reasoning based on the package laws to the low-level one based on our probabilistic relational program logic in §4.2.

### 4.1 Selected rules

Our logic exposes relational judgments of the form  $\models \{pre\} c_0 \sim c_1 \{post\}$ , for which a basic intuition is provided in §2.2. Formally,  $c_0$  and  $c_1$  denote probabilistic stateful code with return type  $A_0$  and  $A_1$  respectively, and the precondition  $m_0 : \text{mem}, m_1 : \text{mem} \vdash pre : \mathbb{P}$  is a proposition with free variables  $m_0$  and  $m_1$  denoting the initial state of the memory (before execution of the code). The postcondition  $m'_0 : \text{mem}, m'_1 : \text{mem} \vdash post : A_0 \times A_1 \rightarrow \mathbb{P}$  is a predicate on the values returned by the executed code, which is parametrized by the variables  $m'_0$  and  $m'_1$  representing the final state of the memory (after execution). The code fragments appearing in a judgment are drawn from the free monad  $\text{code}_{\mathcal{L}, I}$  of §3.1, and meet the further requirement that no oracle calls  $\text{call} \circ \times \text{k}$  appear in them (exactly as in §3.2). The precondition  $pre$  is defined to be a relation between initial memories (for instance,  $m_0 = m_1$ ). Similarly the postcondition  $post$  relates final memories and final results, intuitively obtained after the execution of  $c_i$  on  $m_i$ . We describe how to assign a formal semantics for such probabilistic judgments in §5.2. The semantics is based on the notion of *probabilistic couplings*, already adopted by Barthe et al. [13]. In the remainder of this subsection we describe a selection of our rules.

$$\frac{c : \text{code}_{\mathcal{L}} A}{\models \{m_0 = m_1\} c \sim c \{(a_0, a_1). m'_0 = m'_1 \wedge a_0 = a_1\}} \text{reflexivity}$$

The reflexivity rule relates the code  $c$  to itself when both copies are executed on identical initial memories.

$$\frac{\begin{array}{l} c_0 : \text{code}_{\mathcal{L}_0} A_0 \quad c_1 : \text{code}_{\mathcal{L}_1} A_1 \\ \kappa_0 : A_0 \rightarrow \text{code}_{\mathcal{L}_0} B_0 \quad \kappa_1 : A_1 \rightarrow \text{code}_{\mathcal{L}_1} B_1 \\ \models \{\text{pre}\} c_0 \sim c_1 \{\psi\} \\ \forall a_0 a_1. \models \{\psi(a_0, m_0)(a_1, m_1)\} \kappa_0(a_0) \sim \kappa_1(a_1) \{\text{post}\} \end{array}}{\models \{\text{pre}\} a_0 \leftarrow c_0 ; \kappa_0(a_0) \sim a_1 \leftarrow c_1 ; \kappa_1(a_1) \{\text{post}\}}^{\text{seq}}$$

The seq rule relates two sequentially composed commands using bind by relating each of the sub-commands.

$$\frac{\begin{array}{l} c_0 : \text{code}_{\mathcal{L}} A_0 \quad c_1 : \text{code}_{\mathcal{L}} A_1 \\ \models \{I\} c_0 \sim c_1 \{(a_0, a_1). I \wedge \text{post}(a_0, a_1)\} \\ \models \{I\} c_1 \sim c_0 \{(a_1, a_0). I \wedge \text{post}(a_0, a_1)\} \end{array}}{\models \{I\} c_0 ; c_1 \sim c_1 ; c_0 \{(a_0, a_1). I \wedge \text{post}(a_0, a_1)\}}^{\text{swap}}$$

The swap rule states that if a certain relation on memories  $I$  is invariant with respect to the execution of  $c_0$  and  $c_1$ , then the order in which the commands are executed is not relevant. We used the swap rule in §2.3 in order to swap two independent samplings; in that case the invariant  $I$  consisted in the equality of memories.

$$\frac{\begin{array}{l} c_0, c'_0 : \text{code}_{\mathcal{L}} A_0 \quad c_1 : \text{code}_{\mathcal{L}'} A_1 \\ \models \{\text{pre}\} c_0 \sim c_1 \{\text{post}\} \quad \text{Pr\_code } c_0 = \text{Pr\_code } c'_0 \end{array}}{\models \{\text{pre}\} c'_0 \sim c_1 \{\text{post}\}}^{\text{eqDistrL}}$$

The eqDistrL rule allows us to replace  $c_0$  by  $c'_0$  when both codes have the same denotational semantics as defined by Pr\_code, in the sense of §3.2.

$$\frac{\begin{array}{l} c_0 : \text{code}_{\mathcal{L}} A_0 \quad c_1 : \text{code}_{\mathcal{L}} A_1 \\ \models \{\text{pre}(m_0, m_1)\} c_0 \sim c_1 \{(a_0, a_1). \text{post}(m'_0, a_0)(m'_1, a_1)\} \\ \models \{\text{pre}(m_1, m_0)\} c_1 \sim c_0 \{(a_0, a_1). \text{post}(m'_1, a_1)(m'_0, a_0)\} \end{array}}{\models \{\text{pre}(m_0, m_1)\} c_0 \sim c_1 \{(a_0, a_1). \text{post}(m'_0, a_0)(m'_1, a_1)\}}^{\text{symmetry}}$$

The symmetry rule simply states that the symmetric judgment holds if the arguments of the pre- and postconditions are swapped accordingly.

$$\frac{\begin{array}{l} c_0, c_1 : \mathbb{N} \rightarrow \text{code}_{\mathcal{L}} \text{unit} \quad N : \mathbb{N} \\ \forall i. \models \{I i\} c_0 i \sim c_1 i \{I (i+1)\} \end{array}}{\models \{I 0\} \text{for\_loop } N c_0 \sim \text{for\_loop } N c_1 \{I (N+1)\}}^{\text{for-loop}}$$

The for-loop rule relates two executions of for-loops with the same number of iterations by maintaining a relational invariant through each step of the iteration.

$$\frac{\begin{array}{l} c_0, c_1 : \text{code}_{\mathcal{L}} \text{bool} \quad N : \mathbb{N} \\ \models \{I(\text{true}, \text{true})\} c_0 \sim c_1 \{(b_0, b_1). b_0 = b_1 \wedge I(b_0, b_1)\} \\ \models \{I(\text{true}, \text{true})\} \text{do\_while } N c_0 \sim \\ \text{do\_while } N c_1 \{(b_0, b_1). b_0 = b_1 = \text{false} \vee I(\text{false}, \text{false})\} \end{array}}{\text{do\_while } N c_0 \sim \text{do\_while } N c_1 \{(b_0, b_1). b_0 = b_1 = \text{false} \vee I(\text{false}, \text{false})\}}^{\text{do-while}}$$

The do-while rule relates two bounded while loops with bodies  $c_0$  and  $c_1$ . Every iteration preserves a relational invariant on memories  $I$  that depends on a pair of booleans, and the postcondition also stipulates that  $c_0$  and  $c_1$  return the same boolean, i.e.,  $b_0 = b_1$ . This rule follows the pattern of the unbounded do-while rule defined for simple imperative programs by Maillard et al. [45]. We believe that, with some additional work, their ideas could be used to also support unbounded loops in SSProve.

$$\frac{|A|, |B| < \omega \quad f : A \rightarrow B \text{ bijective}}{\models \{pre\} a \<\$ \text{ uniform } A \sim b \<\$ \text{ uniform } B \{(a, b). f(a) = b \wedge pre\}}^{\text{uniform}}$$

The uniform rule relates sampling from uniform distributions on finite sets  $A$  and  $B$  that are in a bijective correspondence.

$$\frac{D : \text{Op} \quad \sum_{x \in |D|} D(x) = 1 \quad \models \{pre\} c_0 \sim c_1 \{post\} \quad y \notin \text{freevar}(c_0)}{\models \{pre\} y \<\$ D; c_0 \sim c_1 \{post\}}^{\text{dead-sample}}$$

The code  $y \<\$ D; c_0$  samples  $y$  from the subdistribution  $D$ . If  $y$  is never used in  $c_0$ , as indicated by the last premise of the dead-sample rule, then we would like to argue that the sampling constitutes “dead code” and can be ignored. This intuition only holds if  $D$  is a proper distribution rather than a subdistribution. For instance, if  $D$  is the null distribution, the sampling behaves like “assert false” and can certainly not be ignored. The premise  $\sum_{x \in |D|} D(x) = 1$  ensures that  $D$  is indeed a proper distribution (also known as a “lossless subdistribution”). A uniform distribution over a non-empty set would, for instance, constitute a proper distribution in this sense.

$$\frac{D : \text{Op} \quad \sum_{x \in |D|} D(x) = 1 \quad \forall y. \models \{pre\} c_0 y \sim c_1 \{post\}}{\models \{pre\} y \<\$ D; c_0 y \sim c_1 \{post\}}^{\text{sample-irrelevant}}$$

The sample-irrelevant rule has a similar flavor to dead-sample, as it too requires  $D$  to be a proper distribution. We assume that  $c_0 y$  can be related to  $c_1$  for all values of  $y$ . In other words, the choice of a particular value for  $y$  is irrelevant for the pre- and postcondition at hand. Therefore, sampling  $y$  from a proper distribution  $D$  will likewise allow us to conclude that  $c_0 y$  is related to  $c_1$ .

$$\frac{b_0, b_1 : \text{bool}}{\models \{b_0 = b_1\} \text{ assert } b_0 \sim \text{ assert } b_1 \{b_0 = \text{true} \wedge b_1 = \text{true}\}}^{\text{assert}}$$

The assert rule relates two assert commands, as long as “ $b_0 = b_1$ ” holds before the commands. It guarantees “ $b_0 = \text{true} \wedge b_1 = \text{true}$ ” afterwards.

$$\frac{b : \text{bool}}{\models \{b = \text{true}\} \text{ assert } b \sim \text{ return } () \{b = \text{true}\}}^{\text{assertL}}$$

The assertL rule is an asynchronous variant of assert that specifies the behavior of assert, by relating it with return () when the boolean involved in the assert is true. Note that if a code fragment  $c_0$  is shown equivalent to a failure  $\models \{\text{True}\} c_0 \sim \text{ assert false } \{post\}$ , then  $c_0$  must necessarily contain a failure statement as well. Indeed the (sound) model of our program logic, explained in §5, gives rise to a total correctness non-termination semantics: failures only relate to failures.

$$\frac{\begin{array}{l} b_0 : \text{bool} \quad \kappa_0 : b_0 = \text{true} \rightarrow \text{code } A_0 \\ b_1 : \text{bool} \quad \kappa_1 : b_1 = \text{true} \rightarrow \text{code } A_1 \\ \text{pre} \Longrightarrow b_0 = b_1 \\ H_0 : b_0 = \text{true}, H_1 : b_1 = \text{true} \models \{\text{pre}\} \kappa_0 H_0 \sim \kappa_1 H_1 \{\text{post}\} \end{array}}{\models \{\text{pre}\} \text{assert } b_0 \text{ as } h_0 ; \kappa_0 h_0 \sim \text{assert } b_1 \text{ as } h_1 ; \kappa_1 h_1 \{\text{post}\}} \text{assertD}$$

The `assertD` rule allows reasoning about the dependent version of `assert` where the continuation  $\kappa_i$  is only well-defined if the assertion holds, as described in §3.1. As in the `assert` rule, the two assertion conditions  $b_0$  and  $b_1$  may a priori be different. The precondition  $\text{pre}$  has to ensure that both  $b_0$  and  $b_1$  are either both true or both false. The continuations  $\kappa_i$  are defined only in case the assertions succeed. Under this assumption, here represented as the hypotheses  $H_0$  and  $H_1$ , the continuations  $\kappa_i$  must be related for the same  $\text{pre}$  and  $\text{post}$  as the composite statements “`assert`  $b_i$  as  $h_i$ ;  $\kappa_i h_i$ ”.

Following the definition of `assertD`, we can see the intuition for the validity of this rule: if  $b_i$  is true, `assert`  $b_i$  as  $h_i$  is defined as  $\kappa_i h_i$  and we appeal to the last premise. If  $b_i$  is false, both composite statements fail and evaluate to the null distribution.

$$\frac{\begin{array}{l} \ell : \mathcal{L} \quad r : \text{type } \ell \rightarrow \text{code}_{\mathcal{L}} A \quad v : \text{type } \ell \\ \models \{m_0 = m_1\} \text{put } \ell v ; x \leftarrow \text{get } \ell r(x) \sim \\ \text{put } \ell v ; r(v) \{(a_0, a_1). m'_0 = m'_1 \wedge a_0 = a_1\} \end{array}}{\text{put-get}}$$

The `put-get` rule states that looking up the value at location  $\ell$  after storing  $v$  at  $\ell$  results in the value  $v$ . We also have a similar rule to remove a `put` right before another one at the same location, and one for two `get` in a row. More interestingly, we provide asynchronous rules for `get` and `put` which update the pre- or postcondition accordingly.

$$\frac{\begin{array}{l} \ell : \mathcal{L}_0 \quad \kappa : \text{type } \ell \rightarrow \text{code}_{\mathcal{L}_0} A_0 \quad c : \text{code}_{\mathcal{L}_1} A_1 \\ \forall x. \models \{\text{pre} \wedge m_0[\ell] = x\} \kappa(x) \sim c \{\text{post}\} \end{array}}{\models \{\text{pre}\} x \leftarrow \text{get } \ell ; \kappa(x) \sim c \{\text{post}\}} \text{async-get-lhs}$$

$$\frac{\begin{array}{l} \ell : \mathcal{L}_0 \quad \kappa : \text{type } \ell \rightarrow \text{code}_{\mathcal{L}_0} A_0 \quad v : \text{type } \ell \quad c : \text{code}_{\mathcal{L}_1} A_1 \\ \text{pre} \Longrightarrow m_0[\ell] = v \quad \models \{\text{pre}\} \kappa(v) \sim c \{\text{post}\} \end{array}}{\models \{\text{pre}\} x \leftarrow \text{get } \ell ; \kappa(x) \sim c \{\text{post}\}} \text{async-get-lhs-rem}$$

With `async-get-lhs`, one is able to asynchronously read from a memory location and record that information in the precondition. Dually, `async-get-lhs-rem` will recover that information from the precondition. We also use this information which is stored in the preconditions when dealing with memory invariants.

The situation is slightly more complicated for asynchronous writes because writing might break a postcondition. Typically, writing asynchronously when the postcondition ensures that both

memory locations are equal will temporarily break said postcondition.

$$\frac{\ell : \mathcal{L}_0 \quad v : \text{type } \ell \quad c_0 : \text{code}_{\mathcal{L}_0} A_0 \quad c_1 : \text{code}_{\mathcal{L}_1} A_1 \quad \vdash \{ \exists m. \text{pre}(m, m_1) \wedge m_0 = m[\ell \mapsto v] \} \quad c_0 \sim c_1 \{ \text{post} \}}{\vdash \{ \text{pre} \} \text{ put } \ell v ; c_0 \sim c_1 \{ \text{post} \}} \text{async-put-lhs}$$

$$\frac{\forall m_0 m_1. \text{pre}(m_0, m_1) \implies \text{pre}(m_0[\ell \mapsto v], m_1) \quad \vdash \{ \text{pre} \} \quad c_0 \sim c_1 \{ \text{post} \}}{\vdash \{ \exists m. \text{pre}(m, m_1) \wedge m_0 = m[\ell \mapsto v] \} \quad c_0 \sim c_1 \{ \text{post} \}} \text{restore-pre-lhs}$$

async-put-lhs will in fact not modify the postcondition but guarantee that it will hold, provided a modified precondition holds. This new precondition states that the previous precondition  $pre$  was satisfied by a previous memory state, and that the current memory state is the same expect that  $\ell$  now points to  $v$ . If one can show that the precondition is preserved by this memory update, then one can go back to  $pre$  using restore-pre-lhs. In practice, this latter rule is not very useful per se but should be thought of as a simple example of the interaction of writes and invariants. In SSProve we in fact implement a more general rule accounting for any number of writes on both the left- and right-hand sides. Several **put** operations are performed asynchronously, until one can show that the invariant is preserved by all these memory updates.

## 4.2 Proof sketch for Theorem 2.4

If we denote by  $\text{mem}$  the type of memories, then a binary memory predicate

$$m_0 : \text{mem}, m_1 : \text{mem} \vdash \psi : \mathbb{P}$$

holds on a pair of memories  $(h_0, h_1)$ , written  $(h_0, h_1) \models \psi$  if  $\psi [m_0 \mapsto h_0, m_1 \mapsto h_1]$  holds. Moreover, we say that such predicate is *stable* on sets of locations  $\mathcal{L}_0$  and  $\mathcal{L}_1$  if for all  $h_0, h_1$  such that  $(h_0, h_1) \models \psi$ , we have that for all memory locations  $l$ , such that  $l \notin \mathcal{L}_0$  and  $l \notin \mathcal{L}_1$ ,

- (1)  $h_0[l] = h_1[l]$ .
- (2) for all  $v$ ,  $(h_0[l \mapsto v], h_1[l \mapsto v]) \models \psi$ .

When we want to prove that two packages with the same interface are perfectly indistinguishable, we will assume that we have a stable predicate on the locations of the packages, and moreover, that this predicate is an invariant on the different operations of the interface. This invariance of the predicate is the reason why  $\psi$  appears both as a pre- and postcondition in Theorem 2.4. Notice that stable predicates do not impose conditions on the intermediate states of each procedure in the interface of Theorem 2.4, e.g. two related procedures may differ in their internal order of updates, as long as the final results of computations are related.

Before giving the proof sketch for Theorem 2.4, we state a theorem that is also proved in Coq and relates the probabilistic relational program logic with the probabilistic semantics.

**THEOREM 4.1.** *Given values  $a, b$ , if two pieces of code  $c_0, c_1$  are such that*

$$\vdash \{ \psi \} \quad c_0 \sim c_1 \{ (r_0, r_1). \phi(r_0, r_1) \},$$

*$\psi$  holds on the initial memories, and for all  $x, y$  we have that*

$$\phi(x, y) \implies (x = a \iff y = b),$$

*then we have*

$$\Pr[a \leftarrow c_0] = \Pr[b \leftarrow c_1].$$

We are now ready to outline the proof for Theorem 2.4.



PROOF SKETCH OF THEOREM 2.4. We want to prove that for each adversary  $\mathcal{A}$  we have the equality  $\alpha(G_0, G_1)(\mathcal{A}) = 0$ , i.e.,

$$|\Pr[\text{true} \leftarrow \mathcal{A} \circ G_0] - \Pr[\text{true} \leftarrow \mathcal{A} \circ G_1]| = 0.$$

Using the hypothesis and that the predicate  $\psi$  is stable, we perform an induction on the code of the procedure  $\mathcal{A}.\text{Run}$ , to establish

$$\models \{\psi\} (\mathcal{A} \circ G_0).\text{Run}() \sim (\mathcal{A} \circ G_1).\text{Run}() \{(b_0, b_1). b_0 = b_1 \wedge \psi\}.$$

As the induction proceeds, the rules from §4.1 are used to prove each case. We illustrate the get case, which after applying the seq rule with respect to the continuation, and using the inductive hypothesis, reduces to the following judgment:

$$\models \{\psi\} \text{ get } l (\lambda x. \text{return } x) \sim \text{ get } l (\lambda x. \text{return } x) \{(v_0, v_1). v_0 = v_1 \wedge \psi\}$$

As  $\psi$  is stable, we know that the result of get on the left and on the right will coincide (i.e.  $m_0[l] = m_1[l]$ ), because  $l \notin \mathcal{L}_0$  and  $l \notin \mathcal{L}_1$  as  $l$  is a location used in the adversary's code, and we explicitly asked for the adversary memory  $\text{mem}(\mathcal{A})$  to be disjoint from  $\text{mem}(G_0)$  and  $\text{mem}(G_1)$ . As the memory was not changed, the invariant  $\psi$  still holds on the final memory.

As the predicate  $\psi$  holds on the initial memories, and the postcondition  $b_0 = b_1 \wedge \psi$  implies that  $b_0 = \text{true} \iff b_1 = \text{true}$ , we know from Theorem 4.1 that

$$\Pr[\text{true} \leftarrow \mathcal{A} \circ G_0] = \Pr[\text{true} \leftarrow \mathcal{A} \circ G_1],$$

and therefore the advantage is 0.  $\square$

## 5 SEMANTIC MODEL AND SOUNDNESS OF RULES

We build a semantic model validating the rules of the effectful relational program logic from §4. The construction of the model builds upon an effect-modular framework [45], instantiating it with probabilities, simple failures, and global state. We first give in §5.1 an overview of the framework of Maillard et al. [45]. We then informally explain how we apply it in order to (1) obtain a model for a probabilistic relational program logic in §5.2 and (2) enrich it with state in §5.3. The categorical constructions underlying the framework are explained in §5.4, together with the expressive extensions that we need in this work. Finally, in §5.5 we compare this methodology to other approaches for modelling relational program logics appearing in alternative formal cryptography tools, e.g. EasyCrypt and FCF.

### 5.1 Relational effect observation

The aforementioned framework builds upon a monadic representation of effects to provide sound semantics to a large class of relational program logics. As we shall see, this class notably contains logics for reasoning about cryptographic code: code that can manipulate state and sample randomly (see §4.1). A generic relational program logic  $r\mathcal{L}$  is a deductive system with a relational judgment  $\models c_0 \sim c_1 \{w\}$  asserting that pairs of effectful code fragments  $c_0, c_1$  behave according to a given *relational specification*  $w$  connecting the two computations. The exact shape of code and specifications appearing in such a judgment can vary depending on what programming language and logic are considered.

The recipe laid out by Maillard et al. [45] stems from the realization that not only effectful code can be modeled using monads, but specifications can too, and we can build semantics for  $r\mathcal{L}$  using a so-called *relational effect observation* in 3 steps:

- (1) Model the effects involved in the considered left and right programs as monads  $M_0$  and  $M_1$ .
- (2) Turn the collection of relational specifications  $w$  into a *relational specification monad*  $(A_0, A_1) \mapsto W(A_0, A_1)$  ordered by entailment of specifications, written  $w \leq w'$ .

- (3) Finally, find an appropriate *relational effect observation*  $\theta^{A_0 A_1} : M_0 A_0 \times M_1 A_1 \rightarrow W(A_0, A_1)$  mapping a pair of computations in  $M_0 A_0 \times M_1 A_1$  to a relational specification in  $W(A_0, A_1)$ , and preserving the monadic features present on both sides.

Once a relational effect observation  $\theta$  is specified we define a semantic judgment for  $r\mathcal{L}$  as follows:

$$\models_{\theta} c_0 \sim c_1 \{ w \} \iff \theta^{A_0 A_1}(c_0, c_1) \leq w$$

where  $c_i : M_i A_i$  and  $w : W(A_0, A_1)$ .

A typical example of a relational specification monad is the relational backward predicate transformer monad  $\text{BP}(A_0, A_1) := (A_0 \times A_1 \rightarrow \mathbb{P}) \rightarrow \mathbb{P}$ , where  $\mathbb{P}$  is the type of propositions. Intuitively a backward predicate transformer  $w : \text{BP}(A_0, A_1)$  maps a relational postcondition  $\phi$  to a precondition  $w \phi$  *sufficient* to ensure  $\phi$  on the result of the executions of code fragments  $c_0, c_1$  respecting  $w$  (i.e. for which  $\models_{\theta} c_0 \sim c_1 \{ w \}$  for some  $\theta$ ). The preorder on  $\text{BP}(A_0, A_1)$  is given by reverse pointwise implication. For two backward predicate transformers  $w_1, w_2 : \text{BP}(A_0, A_1)$ , we say that  $w_1 \leq w_2$  when  $\forall \phi. w_2 \phi \Rightarrow w_1 \phi$ . Every pre-/postcondition pair  $(pre, post)$  can systematically be translated into a single backward predicate transformer  $\text{toBP}(pre, post)$ :

$$\text{toBP}(pre, post) := \lambda (\phi : A_0 \times A_1 \rightarrow \mathbb{P}). pre \wedge \forall a. post a \Rightarrow \phi a \quad : \text{BP}(A_0, A_1)$$

Note that by looking at the type of BP, it should be clear that BP does not form a monad: it takes two types as input but only returns one. Yet BP somehow still behaves as a monad because we can equip it with bind and return operations satisfying equations akin to the standard monad laws. This is one of the reasons why our precise definitions of relational specification monad and relational effect observation are centered around the notion of *relative monad* instead, explained in detail in §5.4.

## 5.2 Effect observation for probabilities and failures

The technique above can be exploited to build a model for a probabilistic relational program logic. We model probabilistic code using a free monad  $F_{Pr}$  over a probabilistic signature, reusing code  $\mathcal{L}, I$  mentioned in §3.1, where we require that only sampling operations are performed. This code can be assigned a probabilistic semantics using the monad of subdistributions [7, 35], following the track of §3.2, but ignoring considerations around state. This semantics assignment can in fact be seen as a monad morphism  $\delta : F_{Pr} \rightarrow \text{SD}$ .

*Specifications and effect observation.* To model specifications for probabilistic code we use the relational specification monad BP of backward predicate transformers, defined above. The relational effect observation  $\theta_{Pr}$  is based on the notion of probabilistic coupling. A coupling  $d : \text{coupling}(d_0, d_1)$  of two subdistributions  $d_0 : \text{SD}(A_0)$  and  $d_1 : \text{SD}(A_1)$  is a subdistribution over  $A_0 \times A_1$  such that its left and right marginals correspond to  $d_0$  and  $d_1$  respectively. For  $d_i : \text{SD}(A_i)$  two subdistributions we define  $\theta'_{Pr} : \text{SD} \times \text{SD} \rightarrow \text{BP}$  by:

$$\theta'_{Pr}{}^{A_0 A_1}(d_0, d_1) := \lambda (\phi : A_0 \times A_1 \rightarrow \mathbb{P}). \exists (d : \text{coupling}(d_0, d_1)). \forall a_0 a_1. d(a_0, a_1) > 0 \Rightarrow \phi(a_0, a_1).$$

We moreover turn the domain of  $\theta'_{Pr}$  into a product of free monads by setting

$$\theta_{Pr} := \theta'_{Pr} \circ \delta^2 \quad : F_{Pr} \times F_{Pr} \rightarrow \text{BP}.$$

Intuitively, if  $w : \text{BP}(A_0, A_1)$  is obtained out of a  $(pre, post)$  pair, the semantic judgment  $\models_{\theta_{Pr}} c_0 \sim c_1 \{ w \}$  holds when one can find a coupling  $d$  of  $\delta(c_0), \delta(c_1)$  whose support validates *post* whenever *pre* is valid.

Our probabilistic model  $\models_{\theta_{Pr}} c_0 \sim c_1 \{ w \}$  validates state-free accounts of several rules of §4.1. First, since the subdistribution monad is commutative (sampling operations always commute), our semantics validates a state-free variant of the swap rule. Second, as it is often the case for an

arbitrary effect observation, symmetric rules like `uniform` involving similar effectful operations on both sides (here:  $a \leq \$ \text{uniform } A$ ) are validated as well. Third, failing assertions at type  $A$  can be modeled using the null subdistribution on  $A$ , and this interpretation allows us to validate the `assert` rule in our model. Fourth, a state-free variant of the `reflexivity` rule can be established by building, for any subdistribution  $s$ , a coupling  $d : \text{coupling}(s, s)$  of  $s$  with itself. Fifth, any relational effect observation  $\theta$  validates a rule like `seq`. Such a rule is essentially a syntactic formulation of the fact that  $\theta$  should preserve the monadic composition, which is true by definition.

The implementation of the relational effect observation  $\theta_{Pr}$  in Coq depends on on a mathematical theory of couplings and of their interaction with probabilistic programs that we developed. This theory relies internally upon the `mathcomp-analysis` library [2, 3], particularly on their formalization of real numbers, subdistributions and discrete integrals.

### 5.3 Adding state

To extend this first model to stateful code and state-aware specifications, we adapt to our setting the classical notion of *state monad transformer* [40]. A monad transformer maps monads  $M$  to monads  $TM$  and monad morphisms  $\theta$  to monad morphisms  $T\theta$ . In particular, the state monad transformer takes as input a monad  $M$  and a fixed set of states  $S$  and produces a monad with underlying carrier  $\text{StT } M(A) = S \rightarrow M(A \times S)$  with additional ability to read and write elements of  $S$ . Besides, a monad transformer comes equipped with a family of liftings  $\text{lift}^T : \forall M. M \rightarrow TM$  coercing any computation in the original monad  $M$  to a computation in the extended effectful environment  $TM$ . We generalize this to specification monads and build modularly an effect observation  $\theta_{Pr,St}$  on top of  $\theta_{Pr}$ :

$$\theta'_{Pr,St} := \text{StT } \theta_{Pr} \quad : \text{StT}(F_{Pr}^2)(A_0, A_1) \rightarrow \text{StT}(\text{BP})(A_0, A_1)$$

using two sets of global states  $S_0, S_1$  for the left and right, where:

$$\begin{aligned} \text{StT}(F_{Pr}^2)(A_0, A_1) &:= S_0 \times S_1 \rightarrow F_{Pr}(A_0 \times S_0) \times F_{Pr}(A_1 \times S_1), \\ \text{StT}(\text{BP})(A_0, A_1) &:= (A_0 \times S_0 \times A_1 \times S_1 \rightarrow \mathbb{P}) \rightarrow S_0 \times S_1 \rightarrow \mathbb{P}. \end{aligned}$$

Following the definition of  $\theta_{Pr}$  in §5.2, we further extend  $\theta'_{Pr,St}$  by turning its domain into a product of free monads  $F_{Pr,St}^2 := F_{Pr,St} \times F_{Pr,St}$  over a stateful and probabilistic signature. This extension is obtained from  $\theta'_{Pr,St}$  by precomposition with the mapping mentioned in §3.2:

$$\theta_{Pr,St} := \theta'_{Pr,St} \circ \text{Pr\_code}^2.$$

Using the liftings  $\text{lift}^{\text{StT}}$  provided by `StT`, we can build from any purely probabilistic relational judgement  $\models_{\theta_{Pr}} c_0 \sim c_1 \{ w \}$ , a relational judgement  $\models_{\theta_{Pr,St}} c_0 \sim c_1 \{ \text{lift}^{\text{StT}} w \}$  in the state-aware model. This correspondance can be shown to form an embedding of logics: for every  $c_0, c_1, w$  free from state manipulation, derivations of  $\models_{\theta_{Pr,St}} c_0 \sim c_1 \{ \text{lift}^{\text{StT}} w \}$  are in bijective correspondance with derivations of  $\models_{\theta_{Pr}} c_0 \sim c_1 \{ w \}$ . The proof of this latter fact is simplified by the modularity of the construction. This modularity is moreover reflected in the way  $\theta_{Pr,St}(c_0, c_1)$  evaluates. A first pass converts stateful operations of  $c_0, c_1$  and yields state-passing probabilistic code. A second pass interprets the remaining sampling operations and yields state-transforming subdistributions. Lastly a third pass uses  $\theta_{Pr}$  and yields the expected specification  $\theta_{Pr,St}(c_0, c_1) : \text{StT}(\text{BP})(A_0, A_1)$ . The semantic judgment  $\models_{\theta_{Pr,St}} c_0 \sim c_1 \{ w \}$  obtained out of  $\theta_{Pr,St}$  validates all of the rules of our relational program logic (including §4.1).

### 5.4 Categorical foundations of the framework

Our semantics relies on the notion of relational effect observation (§5.1), and on our ability to apply a suitable state transformer to them (§5.3). In this section, we provide categorical definitions

for those notions. Our Coq formalization of the semantics is essentially a formal version of the theory laid out here. Note that Coq types and functions between them form a category that we call `Type`. We will also use the category `PreOrder` of types equipped with a preorder structure (reflexive, transitive relation), and monotone functions.

*Computations and specifications as order-enriched relative monads.* We are interested in modeling probabilistic programs using monads. Yet, in our constructive setting probabilistic computations fail to form a monad. Indeed, our Coq formalization relies on the `mathcomp-analysis` library which defines the type of subdistributions `SD(A)` (see §3.2) only when  $A$  is a “`choiceType`”, that is, a type equipped with an enumeration function for each of its decidable subtypes. This extra choice structure is crucial to define a well-behaved notion of discrete integral on  $A$ , and consequently of subdistribution on  $A$ . Beyond the discrepancy between the domain and codomain of `SD`, it is still possible to endow it with slightly modified versions of the expected `bind` and `return` operations, that satisfy laws comparable to the standard monad laws. Fortunately, Altenkirch et al. [5] explain well how these superficial obstructions due to a mismatch between the domain and codomain of a monad-like structure can be solved using the closely related notion of a *relative monad* instead.

*Definition 5.1 (Relative monad).* Given a functor  $J : \mathcal{I} \rightarrow \mathcal{C}$ , a monad relative to  $J$  (or  $J$ -relative monad) is a functor  $M : \mathcal{I} \rightarrow \mathcal{C}$  equipped with “ $J$ -shifted” return and bind operations

$$\begin{aligned} \text{return} &: \forall (X : \mathcal{I}). \mathcal{C}(JX, MX) \\ \text{bind} &: \forall (X Y : \mathcal{I}). \mathcal{C}(JX, MY) \rightarrow \mathcal{C}(MX, MY) \end{aligned}$$

satisfying  $J$ -shifted versions of the return and bind monad laws.

As a trivial example, any monad  $M : \mathcal{C} \rightarrow \mathcal{C}$  can be seen as a relative monad over the identity functor  $\text{Id}_{\mathcal{C}}$ . Writing `chTy` for the category of choice types (`choiceType`), we are able to package `SD : chTy → Type` as a monad relative to the inclusion functor `chTy → Type` forgetting the extra choice structure. Similarly the probabilistic code monad  $F_{pr}$  must actually be restricted to `chTy` and only forms a relative monad  $F_{pr} : \text{chTy} \rightarrow \text{Type}$  over the inclusion functor.

Regarding specifications, relational specification monads  $W$  fail to form monads as well. Indeed,  $W : \text{Type} \times \text{Type} \rightarrow \text{PreOrder}$  expects two types  $A_0, A_1$  as input but only returns one  $W(A_0, A_1)$ , which is moreover pre-ordered. Again, it turns out that relational specification monads  $W$  (including `BP`) can be seen as relative monads, over the discrete product functor  $\text{dprod} : \text{Type} \times \text{Type} \rightarrow \text{PreOrder} : A_0, A_1 \mapsto A_0 \times A_1$  mapping two types to their product seen as a trivial preorder. Specializing the definition of a relative monad with  $J = \text{dprod}$ , the `bind` and `return` operations of  $W$  take the following form:

$$\begin{aligned} \text{return}^W &: A_0 \times A_1 \rightarrow W(A_0, A_1) \\ \text{bind}^W &: (A_0 \times A_1 \rightarrow W(B_0, B_1)) \rightarrow W(A_0, A_1) \rightarrow W(B_0, B_1) \end{aligned}$$

To soundly model relational program logics, the `bind`<sup>W</sup> operation of the relational specification monad being used should be *monotonic in both arguments*. In our setting, we can in fact easily express that condition by requiring all categorical constructions to be order-enriched [37, 38, 54]. For the sake of readability, we ignore the trivial considerations arising from this enrichment and consider that all the constructions we are dealing with are implicitly order-enriched.

Summing up, in our setting:

- Pairs of computations are modelled by a product  $M_0 \times M_1$  of `Type`-valued (order-enriched) relative monads.
- Specifications are modelled using a relational specification monad  $W$ , i.e., a (order-enriched) relative monad over the discrete product functor  $\text{dprod} : \text{Type} \times \text{Type} \rightarrow \text{PreOrder}$ .

For instance, the domain and codomain of the relational effect observation  $\theta_{Pr}$  defined in §5.2 form respectively a product of Type-valued relative monads, and a relational specification monad.

$$\begin{aligned} \text{dom}(\theta_{Pr}) \text{ is } & F_{Pr} \times F_{Pr} : \text{chTy} \times \text{chTy} \rightarrow \text{Type} \times \text{Type} \\ \text{cod}(\theta_{Pr}) \text{ is } & \text{BP} : \text{Type} \times \text{Type} \rightarrow \text{PreOrder} \end{aligned}$$

*Relational effect observations.* Consider  $M_0, M_1$  two Type-valued relative monads with base functors  $J_0, J_1$  respectively. Let  $W$  be a relational specification monad. The relative monads  $M_0 \times M_1$  and  $W$  organize in the following configuration:

$$\begin{array}{ccc} \text{dom}(M_0) \times \text{dom}(M_1) & \xrightarrow{M_0 \times M_1} & \text{Type}^2 \\ J_0 \times J_1 \downarrow & & \downarrow \text{dprod} \\ \text{Type}^2 & \xrightarrow{W} & \text{PreOrder} \end{array}$$

A relational effect observation  $\theta : M_0 \times M_1 \rightarrow W$  is a collection of mappings

$$\theta^{A_0 A_1} : M_0 A_0 \times M_1 A_1 \rightarrow W(J_0 A_0, J_1 A_1)$$

preserving the bind and return operations of  $M_0, M_1$  up to inequalities:

$$\theta(\text{return}^{M_0} a_0, \text{return}^{M_1} a_1) \leq \text{return}^W(a_0, a_1) \quad (4)$$

$$\theta(\text{bind}^{M_0} f_0 m_0, \text{bind}^{M_1} f_1 m_1) \leq \text{bind}^W(\theta \circ (f_0, f_1)) \theta(m_0, m_1) \quad (5)$$

An instance of relational effect observation is of course given by  $\theta_{Pr}$ . Note that  $\theta_{Pr}$  validates those inequalities but fails to validate them as equalities.

In our development, relational effect observations  $\theta : M_0 \times M_1 \rightarrow W$  are defined as special cases of *lax morphisms between order-enriched relative monads*. We refer the interested reader to our formalization<sup>4</sup> for a precise definition of this notion. In the remainder of this section, we explain how to extend relative monads and lax morphisms between them with state. In particular, this extension will apply to relational effect observations such as  $\theta_{Pr}$ .

*Transforming a relative monad with an appropriate left adjunction.* It is a standard result that every adjunction induces a monad and that every monad is induced by a family of adjunctions (see [43], chapter 6). A similar kind of correspondence holds between left  $J$ -relative adjunctions on one side, and  $J$ -relative monads on the other. The two following definitions appear in [5].

*Definition 5.2 (Left  $J$ -relative adjunction).* Consider functors  $J, L, R$  in the following configuration

$$\begin{array}{ccc} & \mathcal{D} & \\ L \nearrow & & \searrow R \\ \mathcal{I} & \xrightarrow{J} & \mathcal{C} \end{array}$$

We say that  $L$  and  $R$  are  $J$ -relative left and right adjoints respectively ( $L \dashv J \dashv R$ ) if there exists a natural isomorphism  $\forall (X : \mathcal{I}) (Y : \mathcal{C}). \mathcal{D}(LX, Y) \cong \mathcal{C}(JX, RY)$ . In that case, the composition  $RL$  turns out to be a  $J$ -relative monad and is said to be induced by the left relative adjunction  $L \dashv J \dashv R$ .

*Definition 5.3 (Kleisli adjunction of a relative monad).* Let  $M : \mathcal{I} \rightarrow \mathcal{C}$  be a  $J$ -relative monad. We define its Kleisli category  $\text{Kl}(M)$  to have

- as objects, the objects of  $\mathcal{I}$ .
- as morphisms,  $\text{Kl}(M)(X, Y) := \mathcal{C}(JX, MY)$ .

<sup>4</sup><https://github.com/SSProve/ssprove/blob/journal-submission/theories/Relational/OrderEnrichedCategory.v#L379>

It is indeed a category exactly thanks to the monad laws of  $M$ . Moreover there exist functors  $L^M, R^M$  in the following configuration

$$\begin{array}{ccc} & \text{Kl}(M) & \\ L^M \nearrow & & \searrow R^M \\ \mathcal{I} & \xrightarrow{M} & \mathcal{C} \\ & \xleftarrow{J} & \end{array}$$

that form a  $J$ -relative adjunction inducing  $M$ , that is,  $M = R^M L^M$ .

In this work we introduce the following notion.

*Definition 5.4 (Transforming adjunction).* Consider functors  $J, L^b, R$  in the following configuration:

$$\begin{array}{ccc} \mathcal{I} & \xrightarrow{J} & \mathcal{C} \\ L^b \uparrow & & \downarrow R \\ \mathcal{I} & \xrightarrow{J} & \mathcal{C} \end{array}$$

An adjunction  $\alpha : JL^b \dashv R$  is called a transforming adjunction.

If  $\mathcal{I}$  is cartesian,  $\mathcal{C}$  is cartesian closed, and  $J$  preserves cartesian products, the following configuration gives rise to a transforming adjunction  $\sigma : J \circ (- \times S) \dashv S \rightarrow -$ , which we suggestively call “state-transforming adjunction”. Note that the  $J$ -relative monad induced by this adjunction  $X \mapsto S \rightarrow J(X \times S)$  is a  $J$ -shifted version of a standard state monad.

$$\begin{array}{ccc} \mathcal{I} & \xrightarrow{J} & \mathcal{C} \\ - \times S \uparrow & & \downarrow S \rightarrow - \\ \mathcal{I} & \xrightarrow{J} & \mathcal{C} \end{array}$$

**THEOREM 5.5 (RELATIVE TRANSFORMER).** *Given a  $J$ -relative monad  $M : \mathcal{I} \rightarrow \mathcal{C}$  “sitting” on a transforming adjunction  $\alpha : JL^b \dashv R$ , the composition  $RML^b$  is also a  $J$ -relative monad. We call it the relative monad transformed by  $\alpha$  and denote it as  $T_\alpha M$ .*

**PROOF SKETCH.** We can factorize  $M$  through its Kleisli category as shown in Definition 5.3 to obtain  $T_\alpha M := RML^b = RR^M L^M L^b$  and observe that  $L^M L^b \dashv RR^M$ , meaning that  $T_\alpha M$  is the relative monad induced by the latter adjunction.  $\square$

Adding state to a  $J$ -relative monad  $M$  consists in applying the above theorem with the state-transforming adjunction  $\sigma$  defined above to obtain  $\text{StT} M := T_\sigma M$ . In particular, this is how the domain and codomain of  $\text{StT}(\theta_{Pr})$  from §5.3 are defined.

*Transforming lax morphisms.* In order to transform lax morphisms of relative monads (such as  $\theta_{Pr}$ ) we follow the same methodology as in the previous paragraph. Various non-standard categorical notions are at play under the hood: lax morphisms of left relative adjunctions, lax functors, and lax natural transformations. Informally, let  $\theta : M \rightarrow W$  be a lax morphism of relative monads. Let  $\alpha$  be a transforming adjunction for both  $M$  and  $W$ . Then  $\theta$  induces a lax morphism between the Kleisli adjunctions of its domain and codomain:

$$\text{Kl}(\theta) : (L^M \dashv J \dashv R^M) \longrightarrow (L^W \dashv J \dashv R^W)$$

$\text{Kl}(\theta)$  can then be pasted with an appropriate cell to obtain a lax morphism between the transformed adjunctions, which ultimately induces a morphism  $T_\alpha \theta : T_\alpha M \rightarrow T_\alpha W$  between the transformed

relative monads. Adding state to a relational effect observation  $\theta$  now consists in applying the above with the state-transforming adjunction  $\sigma$ . This is how we can obtain  $\text{StT}(\theta_{Pr}) := T_\sigma \theta_{Pr}$  in a modular way.

### 5.5 Comparing approaches to semantic models

We use the semantic framework of Maillard et al. [45] based on effect observations to obtain a formal and foundational approach to relational program logics for cryptographic code. In this section, we compare this methodology to other approaches for modeling relational program logics employed in other formal tools for cryptography, e.g. EasyCrypt and FCF.

The Foundational Cryptography Framework (FCF) [51] develops machine-checked proofs of cryptographic code in the Coq proof assistant. Computations are modeled as elements of a free monad, then interpreted as distributions. This denotational model is subsequently used to derive a program logics using couplings. The approach we take is similar, but we paid special attention to the intermediate monadic structures involved. For instance, oracle calls are implemented in FCF as a specific type of code, whereas we rely on the genericity of free monads at the level of packages. State passing is done explicitly in FCF, while we prefer a more abstract presentation using a state monad transformer. As a result, we obtain a conceptually comfortable decomposition of our computational monads and specifications, at the price of additional work to define the few components that we need for verification of cryptographic code.

Although the implementation of EasyCrypt does not have a proper foundational backend per se, many rules of its probabilistic relational Hoare logic (pRHL) were proved sound with respect to a model in Coq [62], parts of which have been merged into `mathcomp-analysis` [3]. Our own development relies on these very same definitions and lemmas for the probabilistic aspect of the relational specification monads and the underlying theory of couplings. Our contribution here, forced by the organization of the framework from [45], is to show formally that the various lemmas proved in the library indeed build instances of the monadic abstractions that are not explicated in the original development. Amongst the technical difficulties that appeared in that operation, we should mention the fact that distributions are only built-in `mathcomp-analysis` for types equipped with a certain choice structure, reflecting the constructive nature of Coq. However, we cannot endow distributions with such a choice structure, and in particular, distributions do not form monads, but they do form relative monads over the functor forgetting the choice structure (see §5.4). Note that although the definition of the semantic judgment  $\models_{\theta_{Pr, Si}} c_0 \sim c_1 \{ w \}$  can seem abstract at first, it ultimately induces a more direct formulation, such as the one underlying EasyCrypt.

While a direct ad-hoc definition of the model is comparatively simpler to implement, our categorical approach aims to provide more modularity, with the potential to account for multiple effects. This modular approach makes explicit that the model can be restricted to one validating a solely probabilistic program logic (§5.2). Moreover, it should be possible to extend our stateful model with other effects using a similar range of algebraic techniques. However, as things stand now, incorporating new effects in our relational program logic and its associated semantics can only be done on a case by case basis. Developing and using the framework required a high proof effort, in particular to manipulate the various layers of abstractions when proving concrete statements, such as the soundness of rules of the relational program logics specific to the effects involved. Further engineering work would be needed to bring this approach to a competitive level.

## 6 CASE STUDY: KEM-DEM

In order to better demonstrate the practicality of our tool we formalized a more involved public key encryption scheme, KEM-DEM originally proposed by Cramer and Shoup [30], and used for instance in the CryptoBox protocol [21]. KEM-DEM consists in the composition of a key

encapsulation mechanism (KEM) and a data encryption mechanism (DEM), and it can be proved to be indistinguishable from random under chosen ciphertext attacks (IND-CCA), as long as both the KEM and DEM are also IND-CCA schemes.

Our formalization of KEM-DEM showcases high-level SSP arguments that are not present in our previous examples such as parallel composition, the identity package, and the interchange law. Furthermore, we make a more extensive use of our probabilistic relational framework. In particular, we have to account for more interesting invariants than the mere equality of state we were using previously.

Our exposition of KEM-DEM faithfully follows the one by Brzuska et al. [25]. In fact, while conducting the proof in SSProve, we were able to find—in conjunction with the authors of [25]—a flaw in their argument which has led them to propose a revised version of their theorem and its corresponding proof. This section describes the revised proof, which we formalized in SSProve.

### 6.1 The KEY package

The KEM-DEM protocol involves the use of a symmetric key to encrypt the actual data that is going to be sent. The KEY package is used for generating, storing and accessing such a key.

All our statements, as well as the KEY package itself, are parametrized over a type of symmetric keys and distribution  $\text{keyD}$  for generating them. In this respect we generalize [25], which uniformly samples over bitstrings of a given length. We give the KEY package in Figure 14.

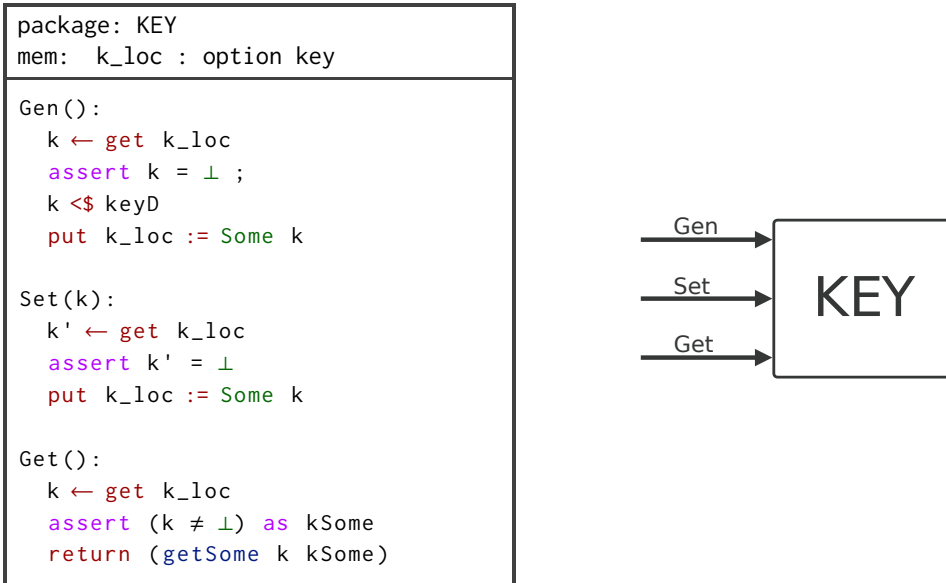


Fig. 14. KEY package

Next we consider packages that may rely on KEY either for storage/generation or for access of an otherwise set key; we will later see the KEM and DEM as instances of those. We call the former keying and the latter keyed games. More formally, a *keying game*  $K$  is given by a core keying game  $CK$  with  $\text{import}(CK^0) = \{\text{Set}\}$ , and  $\text{import}(CK^1) = \{\text{Gen}\}$ , and  $\text{Get} \notin \text{export}(CK^b)$ , while a *keyed game*  $D$  is given by a core keyed game  $CD$  such that  $\text{import}(CD^b) = \{\text{Get}\}$ , and  $\text{Gen} \notin \text{export}(CD^b)$ .



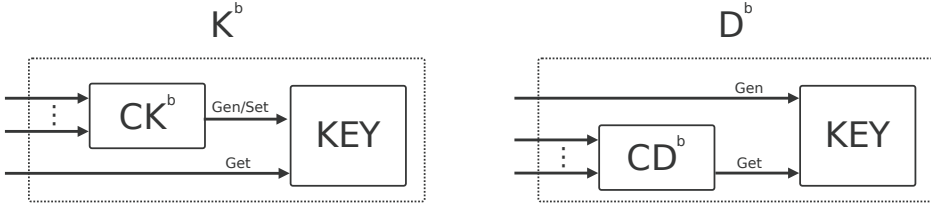


Fig. 15. Keying and keyed games

They are graphically represented in Figure 15 (where we write Gen/Set to indicate that the import is either Gen or Set depending on the secret bit  $b$ ) and defined as follows:

$$\begin{aligned} K^b &= (CK^b \parallel ID_{\{Get\}}) \circ KEY \\ D^b &= (ID_{\{Gen\}} \parallel CD^b) \circ KEY \end{aligned}$$

As we will see, we will define the KEM package as core keying game, and the DEM package as a core keyed game. From that fact alone we will be able to derive security bounds on games that combine them as evidenced by the following lemma.

**LEMMA 6.1 (SINGLE KEY).** *Given a keying game  $K$  and a keyed game  $D$  as above, we have the following inequalities for any distinguisher  $\mathcal{D}$ :*

$$\begin{aligned} &\alpha((CK^0 \parallel CD^0) \circ KEY, (CK^1 \parallel CD^1) \circ KEY)(\mathcal{D}) \leq \\ &\alpha(K)(\mathcal{D} \circ (ID_{\text{export}(CK)} \parallel CD^0)) + \\ &\alpha(D)(\mathcal{D} \circ (CK^1 \parallel ID_{\text{export}(CD)})) \\ \\ &\alpha((CK^0 \parallel CD^0) \circ KEY, (CK^0 \parallel CD^1) \circ KEY)(\mathcal{D}) \leq \\ &\alpha(K)(\mathcal{D} \circ (ID_{\text{export}(CK)} \parallel CD^0)) + \\ &\alpha(D)(\mathcal{D} \circ (CK^1 \parallel ID_{\text{export}(CD)})) + \\ &\alpha(K)(\mathcal{D} \circ (ID_{\text{export}(CK)} \parallel CD^1)) \end{aligned}$$

**PROOF.** We once again make use of Lemma 2.2 and Lemma 2.3 for game-hopping but also of the interchange and identity laws. We will represent the sequence graphically. The packages that we consider are represented in Figure 16 with potentially different instances of  $b$  and  $c$ . For the first inequality we want to relate the figure where  $b = 0$  and  $c = 0$  to the case where  $b = 1$  and  $c = 1$ . To accomplish this we perform the reductions found in Figure 17 which correspond to applications of the identity and interchange laws to the package of Figure 16, and thus they represent equal packages. For instance, we first change  $CK^0$  to  $CK^1$  in  $(CK^0 \parallel CD^0) \circ KEY$  by “pushing”  $CD^0$  to the left (i.e., using the reduction on top), meaning we obtain equal package  $(ID_{\text{export}(CK)} \parallel CD^0) \circ K^0$ . We then hop to  $(ID_{\text{export}(CK)} \parallel CD^0) \circ K^1$ , incurring the term  $\alpha(K)(\mathcal{D} \circ (ID_{\text{export}(CK)} \parallel CD^0))$  in the inequality, and then proceed back to  $(CK^1 \parallel CD^0) \circ KEY$  by doing the inverse of the reduction. The whole proof proceeds in a similar way. The second inequality is a consequence of the first one where we additionally de-idealize the core keying package ( $CK^1$  goes back to  $CK^0$ ), hence the extra term in the inequality.  $\square$

## 6.2 KEM and DEM

In order to be as general as possible, we will assume we are given KEM and DEM schemes. As stated earlier, the KEM will generate a symmetric key and encrypt it using an asymmetric scheme. The DEM will then use that symmetric key to encrypt the data to be sent. To that effect we assume we are given a public and secret key spaces  $pkey$  and  $skey$ , together with a relation  $pkey\_pair$  to

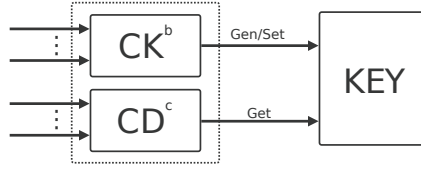


Fig. 16. Keying and keyed games combined

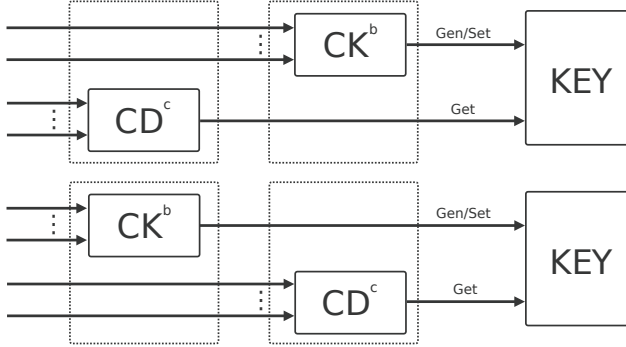


Fig. 17. Reductions for the keying and keyed games

tell which secret key correspond to which public key. We furthermore assume a symmetric key space  $\text{key}$  and an encrypted symmetric key space  $\text{ekey}$  together with distributions  $\text{keyD}$  and  $\text{ekeyD}$  on them.<sup>5</sup> Finally we also assume that we have a type of ciphers and a type of plaintexts together with a distinguished *null* plaintext (which we will write as  $\emptyset$ ), as well as a distribution on ciphers  $\text{cipherD}$ . In [25], these distributions are uniform distributions and these types are described using bitstrings but we decided for a more abstract approach, not only because it is slightly more general, but also because things appear simpler as we do not have to deal with low-level concerns.

A KEM,  $\eta$ , is given by the following:

- (1)  $\eta.\text{kgen}$ , a—state-preserving<sup>6</sup> and typically sampling—procedure which generates a valid public/secret key pair according to the  $\text{pkey\_pair}$  relation;
- (2)  $\eta.\text{encap}$ , a state-preserving procedure which takes a public key  $\text{pk}$  and generates a symmetric key together with its asymmetric encryption with  $\text{pk}$ ;
- (3)  $\eta.\text{decap}$ , a deterministic function—represented by a pure function in Coq—which returns a symmetric key from its encryption and the secret key.

We additionally require that with an appropriate secret key, the original symmetric key can be recovered by applying  $\eta.\text{decap}$  to the encrypted key returned by  $\eta.\text{encap}$ . This specification, and the specification of  $\eta.\text{kgen}$  are handled in our formalisation using the diagonal of our probabilistic relational program logic, i.e., by relating a piece of code to itself.

A DEM,  $\theta$ , is given by the following:

- (1)  $\theta.\text{enc}$ , a deterministic encryption function taking a symmetric key to turn a plaintext into a cipher;
- (2)  $\theta.\text{dec}$ , a deterministic decryption function.

<sup>5</sup>The same  $\text{keyD}$  is used in the  $\text{KEY}$  package.

<sup>6</sup>Observationally, the state must be the same after execution of the procedure.

Note that we do not need to know that  $\theta.\text{dec}$  and  $\theta.\text{enc}$  are inverses of each other to conduct the security proof so we do not require it. Of course, the DEM schemes of interest will verify this property as well.

Using  $\eta$  and  $\theta$  we define the KEM and DEM games in Figure 18. They are then used respectively as core keying and keyed games in the KEM-CCA and DEM-CCA games (represented in Figure 19):

$$\begin{aligned} \text{KEM-CCA}^b &= (\text{KEM}^b \parallel \text{ID}_{\{\text{Get}\}}) \circ \text{KEY} \\ \text{DEM-CCA}^b &= (\text{ID}_{\{\text{Gen}\}} \parallel \text{DEM}^b) \circ \text{KEY} \end{aligned}$$

We finally combine  $\eta$  and  $\theta$  to form a public-key encryption (PKE) in the form the PKE-CCA game

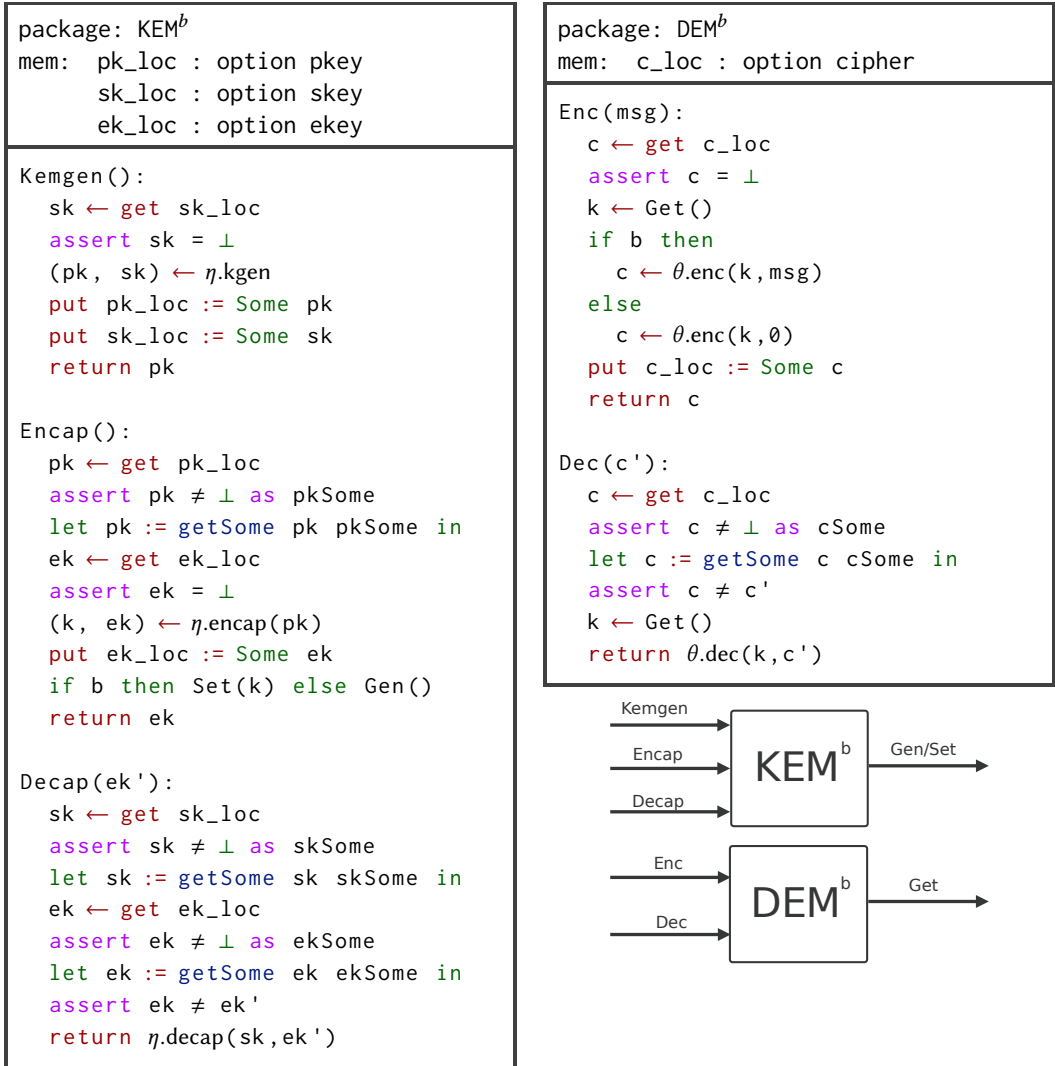


Fig. 18. KEM and DEM games

defined in Figure 20 and Figure 22.

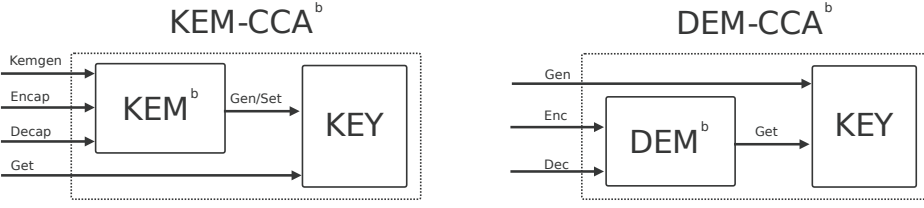


Fig. 19. KEM-CCA and DEM-CCA games

Table 1. Pkgen code comparison of PKE-CCA<sup>b</sup> and AUX<sup>b</sup>

PKE-CCA <sup>b</sup> .Pkgen()	AUX <sup>b</sup> .Pkgen()
<code>sk ← get sk_loc</code>	<code>pk ← get pk_loc</code>
<code>assert sk = ⊥</code>	<code>assert pk = ⊥</code>
<code>(pk, sk) ← η.kgen</code>	<code>sk ← get sk_loc</code>
<code>put pk_loc := Some pk</code>	<code>assert sk = ⊥</code>
<code>put sk_loc := Some sk</code>	<code>(pk, sk) ← η.kgen</code>
<code>return pk</code>	<code>put pk_loc := Some pk</code>
	<code>put sk_loc := Some sk</code>
	<code>return pk</code>

### 6.3 Security of the KEM-DEM construction

To state our PKE security theorem, we also define in Figure 21 and Figure 23 the MOD-CCA package which has the same exports as PKE-CCA, but which will eventually be composed sequentially with KEM, DEM and KEY to form an auxiliary package, featured in Figure 24 and defined as:

$$\text{AUX}^b = \text{MOD-CCA} \circ (\text{KEM}^0 \parallel \text{DEM}^b) \circ \text{KEY}.$$

The security theorem that we formalise is then the following.

**THEOREM 6.2.** *For every adversary<sup>7</sup>  $\mathcal{A}$  we have the following inequality:*

$$\begin{aligned} \alpha(\text{PKE-CCA})(\mathcal{A}) &\leq \\ &\alpha(\text{KEM-CCA})(\mathcal{A} \circ \text{MOD-CCA} \circ (\text{ID}_{\text{export}(\text{KEM})} \parallel \text{DEM}^0)) + \\ &\alpha(\text{DEM-CCA})(\mathcal{A} \circ \text{MOD-CCA} \circ (\text{KEM}^1 \parallel \text{ID}_{\text{export}(\text{DEM})})) + \\ &\alpha(\text{KEM-CCA})(\mathcal{A} \circ \text{MOD-CCA} \circ (\text{ID}_{\text{export}(\text{KEM})} \parallel \text{DEM}^1)) \end{aligned}$$

Before proving this theorem, we will show that PKE-CCA<sup>b</sup> and AUX<sup>b</sup> are perfectly indistinguishable, using Theorem 2.4. After inlining package composition we end up with the code comparison found in Table 1, Table 2, and Table 3. We deliberately add extra newlines to align similar lines of code.

Because only AUX makes use of the KEY package, the k memory location is only used on one side which means that we cannot use equality of heaps as an invariant. Instead, our invariant corresponds to ensuring the following three points:

- (1) equality of heaps on all locations except k\_loc;
- (2) pk\_loc will store ⊥ if and only if sk\_loc stores ⊥;
- (3) whenever k\_loc and ek\_loc are set—i.e., do not contain ⊥—in AUX<sup>b</sup>, ek\_loc will in fact contain the result of the encapsulation of the value stored in k\_loc.

<sup>7</sup>The adversary may not read the state of either PKE-CCA or AUX.

```

package: PKE-CCAb
mem: pk_loc : option pkey
     sk_loc : option skey
     c_loc  : option cipher
     ek_loc : option ekey

Pkgen():
  sk ← get sk_loc
  assert sk = ⊥
  (pk, sk) ← η.kgen
  put pk_loc := Some pk
  put sk_loc := Some sk
  return pk

Pkenc(msg):
  pk ← get pk_loc
  assert pk ≠ ⊥ as pkSome
  let pk := getSome pk pkSome in
  ek ← get ek_loc
  assert ek = ⊥
  c ← get c_loc
  assert c = ⊥
  (k, ek) ← η.encap(pk)
  if b then
    c ← θ.enc(k, msg)
  else
    c ← θ.enc(k, θ)
  put ek_loc := Some ek
  put c_loc := Some c
  return (ek, c)

Pkdec(ek', c'):
  sk ← get sk_loc
  assert sk ≠ ⊥ as skSome
  let sk := getSome sk skSome in
  ek ← get ek_loc
  assert ek ≠ ⊥ as ekSome
  let ek := getSome ek ekSome in
  c ← get c_loc
  assert c ≠ ⊥ as cSome
  let c := getSome c cSome in
  assert (ek, c) ≠ (ek', c')
  k ← η.decap(sk, ek')
  return θ.dec(k, c')

```

Fig. 20. PKE-CCA game

```

package: MOD-CCA
mem: pk_loc : option pkey
     c_loc  : option cipher
     ek_loc : option ekey

Pkgen():
  pk ← get pk_loc
  assert pk = ⊥
  Kemgen()

Pkenc(msg):
  pk ← get pk_loc
  assert pk ≠ ⊥
  ek ← get ek_loc
  assert ek = ⊥
  c ← get c_loc
  assert c = ⊥
  ek ← Encap()
  put ek_loc := Some ek
  c ← Enc(msg)
  put c_loc := Some c
  return (ek, c)

Pkdec(ek', c'):
  pk ← get pk_loc
  assert pk ≠ ⊥
  ek ← get ek_loc
  assert ek ≠ ⊥ as ekSome
  let ek := getSome ek ekSome in
  c ← get c_loc
  assert c ≠ ⊥ as cSome
  let c := getSome c cSome in
  assert (ek, c) ≠ (ek', c')
  if ek = ek' then
    msg ← Dec(c')
  else
    k' ← Decap(ek')
    msg ← θ.dec(k', c')
  return msg

```

Fig. 21. MOD-CCA game



Fig. 22. PKE-CCA game graphically

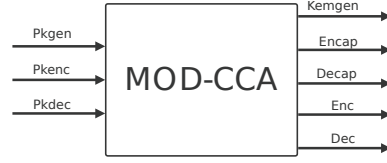


Fig. 23. MOD-CCA games graphically

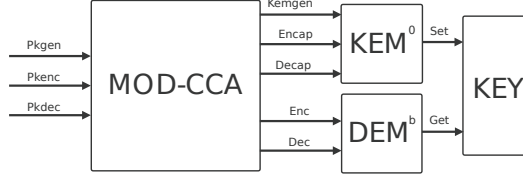


Fig. 24. AUX package construction

To preserve this last invariant we exploit the correctness of the KEM, as we will see later.

We will now address the equivalences corresponding to the three different procedures in the common export interface of  $\text{PKE-CCA}^b$  and  $\text{AUX}^b$ .

*Equivalence for Pkgen.* When looking at Table 1, we can see that the only difference is in the first two lines of  $\text{AUX}^b$  which are absent from  $\text{PKE-CCA}^b$ . Taken in isolation, they would break the equivalence because of the `assert`. Here we can leverage the invariant stating that the locations `pk_loc` and `sk_loc` are always mutually set, so that if `sk_loc` contains  $\perp$  then `pk_loc` does too. To exploit the invariant, we first swap commands on the right-hand side to synchronise with the left-hand side on the read of `sk` and the fact that it must be  $\perp$ . We also verify that the invariant is preserved as `pk_loc` and `sk_loc` are both set at the end of the run.

*Equivalence for Pkenc.* In Table 2 we can see a lot of repetition and locations that are read at different occasions on the left- and right-hand sides. Thankfully our relational program logic supports asynchronous rules for memory reads and writes that *remember* values that have been written and read; they correspond to rules like `async-get-lhs` or `async-put-lhs` that are presented at the end of §4.1. With this we are able to synchronise both sides up to the following line:

$$(k, ek) \leftarrow \eta.\text{encap}(pk)$$

To progress, we cannot merely use a simple application of the `bind` rule because we would then lose the information that `k` and `ek` are related. Instead, we use the specification of  $\eta$  (the KEM) to get as a precondition, for the rest of the comparison, the fact that `ek` is the encryption of `k`. After that, on the right-hand side we make use of the invariant relating `pk`, `k_loc`, and `ek_loc` to ascertain that since `ek_loc` contains  $\perp$ , so must `k_loc`. When the value of `k_loc` is read again on the right-hand side, we proceed as above to *remember* the value that was just stored.

The rest of the proof is straightforward, we only have to show that we preserved the invariant when overwriting the memory, which means that we must show that the newly stored values in `k_loc` and `ek_loc` must indeed correspond to a pair of a key and its encryption, a fact that we recovered above.

*Equivalence for Pkdec.* For the most part before the `if` in Table 3, the equivalence proof is conducted in roughly the same way as above. Then we proceed with a case-analysis on `ek = ek'`. In the `else` branch, all of the asserts hold, as they hold in the lines above and the invariant stating

Table 2. Pkenc code comparison of  $\text{PKE-CCA}^b$  and  $\text{AUX}^b$ 

$\text{PKE-CCA}^b$ . Pkenc(msg)	$\text{AUX}^b$ . Pkenc(msg)
<pre> pk ← get pk_loc assert pk ≠ ⊥ as pkSome let pk := getSome pk pkSome in ek ← get ek_loc assert ek = ⊥ c ← get c_loc assert c = ⊥  (k, ek) ← η.encap(pk)  if b then   c ← θ.enc(k, msg) else   c ← θ.enc(k, θ) put ek_loc := Some ek put c_loc := Some c return (ek, c) </pre>	<pre> pk ← get pk_loc assert pk ≠ ⊥  ek ← get ek_loc assert ek = ⊥ c ← get c_loc assert c = ⊥ pk ← get pk_loc assert pk ≠ ⊥ as pkSome let pk := getSome pk pkSome in ek ← get ek_loc assert ek = ⊥ (k, ek) ← η.encap(pk) put ek_loc := Some ek k' ← get k_loc assert k' = ⊥ put k_loc := Some k put ek_loc := Some ek c ← get c_loc assert c = ⊥ k ← get k_loc assert k ≠ ⊥ as kSome let k := getSome k kSome in if b then   c ← θ.enc(k, msg) else   c ← θ.enc(k, θ) put c_loc := Some c put c_loc := Some c return (ek, c) </pre>

that  $\text{pk\_loc}$  is  $\perp$  if and only if  $\text{sk\_loc}$  is satisfied, and furthermore the case-analysis yielded  $\text{ek} \neq \text{ek}'$ . The rest of the code in the `else` branch then goes on to produce exactly the same result as the left-hand side.

The more interesting bit happens in the `then` branch where there is no call to the decapsulation procedure  $\eta.\text{decap}$  of the KEM. Instead, we exploit the invariant that states that the stored encrypted key in  $\text{ek\_loc}$  corresponds to the encryption of the key in  $\text{k\_loc}$  using the public key in  $\text{pk\_loc}$ , a fact which we encoded by saying that in this case  $\text{k}$  is equal to  $\eta.\text{encap}(\text{sk}, \text{ek})$ . We conclude remembering that we are in the branch where  $\text{ek} = \text{ek}'$ .

Now that the three procedures have been shown equivalent, we know that the two packages are indeed perfectly indistinguishable. We can thus proceed to the proof of Theorem 6.2.

Table 3. Pkdec code comparison of  $\text{PKE-CCA}^b$  and  $\text{AUX}^b$ 

$\text{PKE-CCA}^b.\text{Pkdec}(ek', c')$	$\text{AUX}^b.\text{Pkdec}(ek', c')$
<pre> sk ← get sk_loc assert sk ≠ ⊥ as skSome let sk := getSome sk skSome in ek ← get ek_loc assert ek ≠ ⊥ as ekSome let ek := getSome ek ekSome in c ← get c_loc assert c ≠ ⊥ as cSome let c := getSome c cSome in assert (ek, c) ≠ (ek', c')  k ← η.decrap(sk, ek') return θ.dec(k, c') </pre>	<pre> pk ← get pk_loc assert pk ≠ ⊥  ek ← get ek_loc assert ek ≠ ⊥ as ekSome let ek := getSome ek ekSome in c ← get c_loc assert c ≠ ⊥ as cSome let c := getSome c cSome in assert (ek, c) ≠ (ek', c') if ek = ek' then   c ← get c_loc   assert c ≠ ⊥ as cSome   let c := getSome c cSome in   assert c ≠ c'   k ← get k_loc   assert k ≠ ⊥ as kSome   let k := getSome k kSome in   msg ← θ.dec(k, c') else   sk ← get sk_loc   assert sk ≠ ⊥ as skSome   let sk := getSome sk skSome in   ek ← get ek_loc   assert ek ≠ ⊥ as ekSome   let ek := getSome ek ekSome in   assert ek ≠ ek'   k' ← η.decrap(sk, ek')   msg ← θ.dec(k', c') return msg </pre>

PROOF OF THEOREM 6.2. We use Lemma 2.2 to obtain

$$\begin{aligned}
\alpha(\text{PKE-CCA})(\mathcal{A}) &\leq \\
&\alpha(\text{PKE-CCA}^1, \text{MOD-CCA} \circ (\text{KEM}^0 \parallel \text{DEM}^1) \circ \text{KEY})(\mathcal{A}) + \\
&\alpha(\text{MOD-CCA} \circ (\text{KEM}^0 \parallel \text{DEM}^1) \circ \text{KEY}, \text{MOD-CCA} \circ (\text{KEM}^0 \parallel \text{DEM}^0) \circ \text{KEY})(\mathcal{A}) + \\
&\alpha(\text{MOD-CCA} \circ (\text{KEM}^0 \parallel \text{DEM}^0) \circ \text{KEY}, \text{PKE-CCA}^0)(\mathcal{A})
\end{aligned}$$

which corresponds to the following if we fold AUX:

$$\begin{aligned}
\alpha(\text{PKE-CCA})(\mathcal{A}) &\leq \\
&\alpha(\text{PKE-CCA}^1, \text{AUX}^1)(\mathcal{A}) + \\
&\alpha(\text{MOD-CCA} \circ (\text{KEM}^0 \parallel \text{DEM}^1) \circ \text{KEY}, \text{MOD-CCA} \circ (\text{KEM}^0 \parallel \text{DEM}^0) \circ \text{KEY})(\mathcal{A}) + \\
&\alpha(\text{AUX}^0, \text{PKE-CCA}^0)(\mathcal{A})
\end{aligned}$$



Since  $\text{PKE-CCA}^b$  is perfectly indistinguishable from  $\text{AUX}^b$  we are left with

$$\alpha(\text{PKE-CCA})(\mathcal{A}) \leq \alpha(\text{MOD-CCA} \circ (\text{KEM}^0 \parallel \text{DEM}^1) \circ \text{KEY}, \text{MOD-CCA} \circ (\text{KEM}^0 \parallel \text{DEM}^0) \circ \text{KEY})(\mathcal{A})$$

Using symmetry and Lemma 2.3 we replace the second quantity with

$$\alpha((\text{KEM}^0 \parallel \text{DEM}^0) \circ \text{KEY}, (\text{KEM}^0 \parallel \text{DEM}^1) \circ \text{KEY})(\mathcal{A} \circ \text{MOD-CCA})$$

which corresponds to an instance of the left-hand side of the second inequality of Lemma 6.1 using  $\mathcal{A} \circ \text{MOD-CCA}$  as distinguisher, meaning it is smaller than

$$\begin{aligned} & \alpha(\text{KEM-CCA})(\mathcal{A} \circ \text{MOD-CCA} \circ (\text{ID}_{\text{export}(\text{KEM})} \parallel \text{DEM}^0)) + \\ & \alpha(\text{DEM-CCA})(\mathcal{A} \circ \text{MOD-CCA} \circ (\text{KEM}^1 \parallel \text{ID}_{\text{export}(\text{DEM})})) + \\ & \alpha(\text{KEM-CCA})(\mathcal{A} \circ \text{MOD-CCA} \circ (\text{ID}_{\text{export}(\text{KEM})} \parallel \text{DEM}^1)) \end{aligned}$$

We thus conclude using transitivity.  $\square$

## 7 CASE STUDY: $\Sigma$ -PROTOCOLS

$\Sigma$ -protocols form an important class of zero-knowledge protocols [31]. A  $\Sigma$ -protocol is defined on a relation  $\mathbf{R}$  for which it proves zero-knowledge in the presence of an honest verifier. In other words, it allows a prover to convince a verifier that it knows a witness  $w$  and a public statement  $h$  such that  $(w, h) \in \mathbf{R}$ . The prover can do so without revealing any information about the witness.

In this section, we show how we can define the class of  $\Sigma$ -protocols in SSProve. We then prove security of a transformation converting a  $\Sigma$ -protocol in our class of protocols into a commitment scheme. A commitment scheme is a cryptographic primitive allowing anyone to publicly commit themselves to a value without revealing the value itself. Moreover, the party committing to the message can freely reveal the message at a later time with the guarantee that the value revealed is the value publicly committed to earlier.

Finally, we conclude the section by proving Schnorr's protocol [58] to be a member of the class of  $\Sigma$ -protocols and prove concrete security bounds. Schnorr's protocol allows a prover to convince a verifier that it knows the discrete logarithm of a group element.

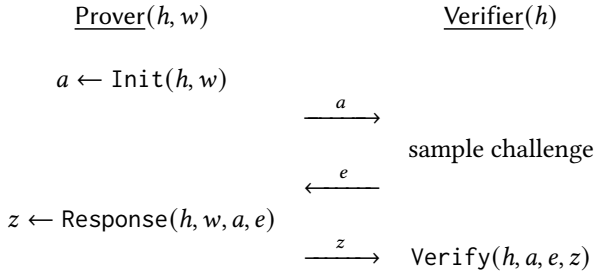


Fig. 25.  $\Sigma$ -Protocol overview

The general flow of a  $\Sigma$ -protocol can be seen in Figure 25. First, the Prover uses the secret information to compute a message which is sent to the Verifier. Second, the Verifier samples a challenge uniformly at random from some challenge space and sends it to the Prover. Third, the Prover computes a response based on the secret information, the message, and the challenge. The response is then sent to the Verifier. Finally, the Verifier takes the public information, message, challenge, and response and checks if it is convinced that the Prover knows the secret.

The combination of the message, challenge, and response is commonly referred to as the *transcript* of the protocol.

We say that a  $\Sigma$ -protocol is secure when both of the following hold.

- (1) There exists an efficient simulator which, given the public information and a fixed challenge, can produce a transcript that is indistinguishable from a real execution of the protocol with the same challenge. This is commonly referred to as the protocol being *special honest-verifier zero-knowledge*.
- (2) Given two accepting transcripts with the same initial message and different challenges, the witness for the relation can be reconstructed. This property is known as *special soundness*.

## 7.1 The SIGMA package

For the most general representation, we assume that we are given access to a SIGMA scheme. This scheme is given by the following:

- (1) *Init*, a stateful procedure generating a message from the witness and public statement.
- (2) *Response*, a stateful procedure generating a response from the witness, public statement, previous message, and any challenge.
- (3) *Verify*, a deterministic procedure returning a boolean based on all information sent in the protocol.
- (4) *Simulate*, a function computing a tuple (message, response) from a public statement and any challenge.
- (5) *Extract*, a procedure that given two transcripts either outputs a witness or fails.

Security is then defined as several games interacting with the SIGMA scheme. The various security games are shown in Figure 27 and Figure 26.

*Definition 7.1.* For any instantiation of the SIGMA scheme and any adversary  $\mathcal{A}$  we say that it is Special Honest-Verifier Zero-Knowledge with advantage  $\epsilon_{SHVZK}$ , if  $\alpha(\text{EXEC})(\mathcal{A}) \leq \epsilon_{SHVZK}$ .

*Definition 7.2.* Let  $\mathcal{P}$  be any oracle with a single procedure Gen, which takes as input the public part of the relation and outputs two transcripts with the same initial message  $A$ . For any instantiation of the SIGMA scheme and any adversary  $\mathcal{A}$  we say that it has Special Soundness advantage,  $\epsilon_{SOUND}$ , if  $\alpha(\text{SOUND} \circ \mathcal{P})(\mathcal{A}) \leq \epsilon_{SOUND}$ .

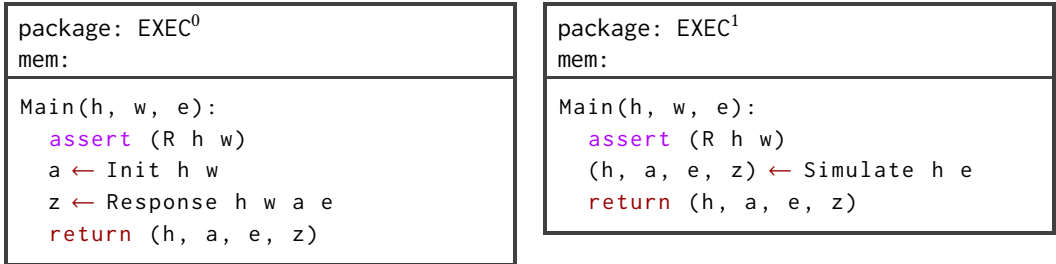


Fig. 26. EXEC game

## 7.2 Commitment Schemes from $\Sigma$ -Protocols

A commitment scheme is another cryptographic primitive allowing a committer with some message  $msg$  to convince a verifier of two things: First, that  $msg$  has a fixed value set before contacting the verifier. Second, that the committer can at any later time reveal the value of  $msg$  to the verifier.

<pre> package: SOUND<sup>0</sup> mem:  Extract(h):   (a, (e, z), (e', z')) ← P.Gen(h   )   v ← Verify h a e z   v' ← Verify h a e' z'   if ((e ≠ e') &amp;&amp; v &amp;&amp; v') then     w' ← Extract h a e e' z z'   return R h w' </pre>	<pre> package: SOUND<sup>1</sup> mem:  Extract(h):   (a, (e, z), (e', z')) ← P.Gen(h   )   v ← Verify h a e z   v' ← Verify h a e' z'   return (e ≠ e') &amp;&amp; v &amp;&amp; v' </pre>
---	---

Fig. 27. SOUND game

In particular, it must be the case that the verifier is convinced that the revealed message has not been changed from the original fixed message.

A commitment scheme is parametrized by types of messages, opening keys, and commitments. The scheme is given by the following:

- (1) *Commit*, a probabilistic and stateful procedure, which produces a commitment from a message and an opening key.
- (2) *Open*, a stateful procedure, which outputs a message and opening key.
- (3) *Ver*, which takes as input a commitment, message, and opening key and checks the validity of the commitment.

*Definition 7.3.* A commitment scheme is called *secure* when it is both *hiding* and *binding*:

- *Hiding*: For any commitment  $c$  produced from message  $msg$  there exists a message  $msg' \neq msg$  with commitment  $c'$  indistinguishable from  $c$ .
- *Binding*: For any commitment  $c$  it is infeasible to find messages with openings keys  $(msg, o)$  and  $(msg', o')$  with  $msg \neq msg'$  such that both messages are valid openings for  $c$ .

For any given instance of a commitment scheme we define the security definitions from Definition 7.3 as the security games seen in Figure 29 and Figure 30.

Following the presentation of [31], we show how our SIGMA scheme with related security games can be used to construct a commitment scheme. The key component of this transformation is the COM package seen in Figure 28. Here, COM depends on the SIGMA scheme and its respective EXEC package. COM then exports three procedures:

- (1) *Commit*, which uses the public and secret parts of the relation to produce a commitment to the challenge  $e$ . For this transformation, the commitment is the initial message of the underlying  $\Sigma$ -protocol.
- (2) *Open*, which returns the required information to verify the commitment. In our case, this constitutes the challenge  $e$  and the response of the  $\Sigma$ -protocol.
- (3) *Ver*, which takes the commitment and the opening information and verifies their consistency. This, again, is dependent on the underlying  $\Sigma$ -protocol.

In this transformation, the types of the underlying  $\Sigma$ -protocol dictate the types of the commitment scheme. In particular, the message type of the commitment scheme is the type of the challenge used in the  $\Sigma$ -protocol.

First, in Theorem 7.4 the construction is shown to be hiding with security bounded by the underlying  $\Sigma$ -protocol.

To show binding we reduce the problem to that of finding the witness for the relation of the  $\Sigma$ -protocol. Let  $\mathcal{P}$  be an oracle with a single procedure:  $\mathcal{P}.\text{Gen}$ , which takes as input the public part of the relation and produces a commitment and two openings. Breaking the binding property implies that there must exist an instantiation of  $\mathcal{P}$ , such that two openings are both valid and distinct. Considering the case where  $\mathcal{P}$  does indeed break binding, we then show in Theorem 7.5 that the probability of breaking the binding property can be bounded by the probability of guessing the witness for the relation of the underlying  $\Sigma$ -protocol.

```

package: COM
mem: e_loc : option challenge
     z_loc : option response

Commit(h, w, e):
  (h, a, e, z) ← Run(h, w, e)
  put e_loc := Some e
  put z_loc := Some z
  return a

Open():
  e ← get e_loc
  z ← get z_loc
  assert (e ≠ ⊥) as eSome
  assert (z ≠ ⊥) as zSome
  return (getSome e eSome, getSome z zSome)

Ver(h, a, e, z):
  v ← Verify(h, a, e, z)
  return v

```

Fig. 28. COM Package

**THEOREM 7.4.** *Assuming the underlying SIGMA scheme is Special Honest-Verifier Zero-Knowledge with advantage  $\forall \mathcal{B}, \alpha(\text{EXEC})(\mathcal{B}) \leq \epsilon_{\text{SHVZK}}$  then for every adversary  $\mathcal{A}$  the following inequality holds:*

$$\begin{aligned} & \alpha(\text{HIDE} \circ \text{COM} \circ \text{EXEC}^1)(\mathcal{A}) \leq \\ & \alpha(\text{HIDE}^0 \circ \text{COMEXEC}^0, \text{HIDE}^1 \circ \text{COM} \circ \text{EXEC}^0)(\mathcal{A}) + \\ & 2 \cdot \epsilon_{\text{SHVZK}} \end{aligned}$$

**PROOF OF THEOREM 7.4.** Applying Lemma 2.2 we obtain:

$$\begin{aligned} & \alpha(\text{HIDE}^0 \circ \text{COM} \circ \text{EXEC}^1, \text{HIDE}^1 \circ \text{COM} \circ \text{EXEC}^1)(\mathcal{A}) \leq \\ & \alpha(\text{HIDE}^0 \circ \text{COM} \circ \text{EXEC}^1, \text{HIDE}^0 \circ \text{COM} \circ \text{EXEC}^0)(\mathcal{A}) + \\ & \alpha(\text{HIDE}^0 \circ \text{COM} \circ \text{EXEC}^0, \text{HIDE}^1 \circ \text{COM} \circ \text{EXEC}^0)(\mathcal{A}) + \\ & \alpha(\text{HIDE}^1 \circ \text{COM} \circ \text{EXEC}^0, \text{HIDE}^1 \circ \text{COM} \circ \text{EXEC}^1)(\mathcal{A}) \end{aligned}$$

Using our assumption of  $\forall (B), \alpha(\text{EXEC})(\mathcal{B}) \leq \epsilon_{\text{SHVZK}}$  we can simplify the equation further to:

```

package: HIDEb
mem:
Hide(h, w, e):
  if b then
    c ← Commit(h, w, e)
    return c
  else
    e <$ challenged
    c ← Commit(h, w, e)
    return c

```

Fig. 29. HIDE Package

```

package: BIND
mem:
Extract(h):
  (c, o, o') ← P.Gen(h)
  v ← Verify h a e z
  v' ← Verify h a e' z'
  return (o ≠ o') && v && v'

```

Fig. 30. BIND game

$$\begin{aligned}
& \alpha(\text{HIDE}^0 \circ \text{COM} \circ \text{EXEC}^1, \text{HIDE}^1 \circ \text{COM} \circ \text{EXEC}^1)(\mathcal{A}) \leq \\
& \epsilon_{\text{SHVZK}} + \\
& \alpha(\text{HIDE}^0 \circ \text{COM} \circ \text{EXEC}^0, \text{HIDE}^1 \circ \text{COM} \circ \text{EXEC}^0)(\mathcal{A}) + \\
& \epsilon_{\text{SHVZK}}
\end{aligned}$$

□

**THEOREM 7.5.** *Assuming the underlying SIGMA scheme has special soundness with advantage  $\forall \mathcal{B}, \alpha(\text{SOUND})(\mathcal{B}) \leq \epsilon_{\text{BIND}}$  then for every adversary  $\mathcal{A}$ , and commitment producing oracle  $\mathcal{P}$  producing commitments, we have the following inequality:*

$$\alpha(\text{BIND} \circ \mathcal{P}, \text{SOUND}^0 \circ \mathcal{P})(\mathcal{A}) \leq \epsilon_{\text{BIND}}$$

To prove Theorem 7.5, we use Lemma 2.2 and Lemma 2.3 to replace the commitments with  $\Sigma$ -protocol transcripts, via  $\text{SOUND}^1$ .

**7.2.1 Formalisation in CryptHOL.** The construction deriving commitment schemes from  $\Sigma$ -protocols has also been formalized in CryptHOL [26]. In their definition, the commitment scheme construction is dependent on the types needed to instantiate a  $\Sigma$ -protocol. The proofs of hiding and binding are then quantified over any secure  $\Sigma$ -protocol that can be formed from the specified types. In particular, it is required that the distinguishing advantage on the special honest-verifier zero-knowledge security game is 0. Moreover, they assume that the *Response* and *Verify* procedures of the  $\Sigma$ -protocol terminate on all inputs.

Contrasting to our security bounds defined in Theorem 7.4 and Theorem 7.5 we make no assumptions on the underlying  $\Sigma$ -protocol other than they implement the SIGMA interface. Because our results hold for any SIGMA package, we obtain a more general notion of security for the hiding property in Theorem 7.4, where the security bound is directly related to the security of the underlying  $\Sigma$ -protocol.

The imperfect game hops are justified by the package laws of SSProve, which can be seen in the proof of Theorem 7.4. In particular, this allows us to accumulate the advantage from our game-hops into our final security bound, whatever the respective intermediate advantages may be.

Without the ability to reason about the advantage of composed packages the proof would have had to involve significantly more steps, or make the same assumptions as in [26]. Namely, if we adopt the same assumption, then the proof of the security bounds can be done entirely within

```

package: SCHNORR
mem: r : option secret

Init(h, w):
  r <$ secretD
  return gr

Response(h, w, a, e):
  return r + e * w

Verify(h, w, a, e, z):
  return gz = a * he

Simulator(h, e):
  z <$ secretD
  return (h, gz * h-e, e, z)

Extractor(h, a, e, e', z, z'):
  return (z - z') / (e - e').

```

Fig. 31. SCHNORR package

the relational logic itself. More concretely, the assumption of perfect special honest verifier zero-knowledge reduces the statement from an adversary comparing both programs to the two programs being equivalent in the relation logic.

### 7.3 Concrete Implementation: Schnorr's protocol

The Schnorr protocol [58] is parametrized over a multiplicative cyclic group  $(\mathcal{G}, *)$  with  $q$  elements generated by  $g$ . Schnorr's protocol is a  $\Sigma$ -protocol for the relation  $(h, w) \in \mathbf{R} \iff h = g^w$ , where  $w$  is an element of  $\mathbb{Z}_q$  and  $h \in \mathcal{G}$ .

Messages are elements  $a \in \mathcal{G}$  and responses are elements  $z \in \mathcal{G}$ . Challenges are sampled from a uniform distribution over  $\mathbb{Z}_q$ .

The protocol is implemented as an SSP package as shown in Figure 31. In particular, the package exports match the expected imports of our  $\Sigma$ -protocol security statements.

LEMMA 7.6 (SCHNORR SHVZK). *For any adversary  $\mathcal{A}$  we have the following equality:*

$$\alpha(\text{EXEC})(\mathcal{A}) = 0$$

PROOF OF LEMMA 7.6. We use Theorem 2.4 to show the two packages are perfectly indistinguishable. After inlining the definition of Schnorr's protocol we get the code found in Table 4. Since the EXEC package does not use state and Schnorr's protocol only uses state for its internal  $r$  variable we can use equality of heaps as our invariant.

When comparing the code in Table 4 the programs immediately differ in the use of their randomly sampled values. To make the programs agree on return values we use the `uniform` rule from §4.1. First, we see that both sides assert that the relation  $\mathbf{R}$  holds. We remove both asserts and assume for the rest of the proof that  $\mathbf{R}$  holds. Applying `uniform` with  $f : x \mapsto x + e * w$  and moving all

Table 4. Code comparison of EXEC<sup>0</sup> and EXEC<sup>1</sup> in Schnorr's protocol

EXEC <sup>0</sup> .Run(h, w, e)	EXEC <sup>1</sup> .Run(h, w, e)
<code>assert h = g<sup>w</sup></code>	<code>assert h = g<sup>w</sup></code>
<code>r &lt;\$ secretD</code>	<code>z &lt;\$ secretD</code>
<code>a ← g<sup>r</sup></code>	<code>a ← g<sup>z</sup> * h<sup>-e</sup></code>
<code>z ← r + e * w</code>	
<code>return (h, a, e, z)</code>	<code>return (h, a, e, z)</code>

Table 5. Code comparison of SOUND<sup>0</sup> ◦ P and SOUND<sup>1</sup> ◦ P in Schnorr's protocol

(SOUND <sup>0</sup> ◦ P).Extract(h)	(SOUND <sup>1</sup> ◦ P).Extract(h)
<code>(a, (e, z), (e', z')) ← P.Gen(h)</code>	<code>(a, (e, z), (e', z')) ← P.Gen(h)</code>
<code>v ← g<sup>z</sup> = a * h<sup>e</sup></code>	<code>v ← g<sup>z</sup> = a * h<sup>e</sup></code>
<code>v' ← g<sup>z'</sup> = a * h<sup>e'</sup></code>	<code>v' ← g<sup>z'</sup> = a * h<sup>e'</sup></code>
<code>if ((e ≠ e') &amp;&amp; v &amp;&amp; v') then</code>	
<code>w' ← (z - z') / (e - e')</code>	
<code>return (h = g<sup>w'</sup>)</code>	
<code>else</code>	
<code>return false</code>	<code>return ((e ≠ e') &amp;&amp; v &amp;&amp; v')</code>

constants into the return statement we obtain:

$$\begin{aligned}
(g^r, e, r + e \cdot w) &= (g^{f(r)} \cdot h^{-e}, e, f(r)) \\
&= (g^{r+e \cdot w} \cdot h^{-e}, e, r + e \cdot w) \\
&= (g^r \cdot h^e \cdot h^{-e}, e, r + e \cdot w) \\
&= (g^r, e, r + e \cdot w),
\end{aligned}$$

where we use the fact that the relation  $h = g^w$  holds for our particular values of  $h$  and  $w$ .  $\square$

LEMMA 7.7 (SCHNORR SPECIAL-SOUNDNESS). *For any adversary  $\mathcal{A}$  and for all transcript producing programs  $\mathcal{P}$  we have the following equality:*

$$\alpha(\text{SOUND} \circ \mathcal{P})(\mathcal{A}) = 0$$

PROOF OF LEMMA 7.7. We use Theorem 2.4 to show the two packages are perfectly indistinguishable. After inlining the definition of Schnorr's protocol we get the code found in Table 5. Since neither Schnorr's protocol nor the SOUND package use state, we can use equality of heaps as our invariant. Observing the code in Table 5 the same  $\mathcal{P}$  package is used on both sides. These are indistinguishable if  $h = g^{w'}$ . We show this indeed holds:

$$g^z = a \cdot h^e \wedge g^{z'} = a \cdot h^{e'} \wedge e \neq e' \implies h = g^{(z-z')/(e-e')} = g^{w'}.$$

$\square$

Based on Lemma 7.6 and Lemma 7.7 we can instantiate Theorem 7.4 and Theorem 7.5. For the latter, we can directly apply the theorem to show that any adversary has no advantage between

Table 6. Code comparison of  $\text{HIDE}^0 \circ \text{COM} \circ \text{EXEC}^0$  and  $\text{HIDE}^1 \circ \text{COM} \circ \text{EXEC}^0$ 

$(\text{HIDE}^0 \circ \text{COM} \circ \text{EXEC}^0).\text{Hide}(h, w, e)$	$(\text{HIDE}^1 \circ \text{COM} \circ \text{EXEC}^0).\text{Hide}(h, w, e)$
<pre> assert (R h w) r &lt;\$ secretD a ← g<sup>r</sup> z ← r + e * w put e_loc := Some e put z_loc := Some z return a </pre>	<pre> e &lt;\$ challenged assert (R h w) r &lt;\$ secretD a ← g<sup>r</sup> z ← r + e * w put e_loc := Some e put z_loc := Some z return a </pre>

the binding game and directly extracting the witness. For the former, the adversary also has no advantage, which we show in Theorem 7.8.

**THEOREM 7.8.** *For every adversary  $\mathcal{A}$  we get the following equality for the commitment scheme instantiated from Schnorr's protocol:*

$$\alpha(\text{HIDE} \circ \text{COM} \circ \text{EXEC}^1)(\mathcal{A}) = 0$$

**PROOF OF THEOREM 7.8.** We use Theorem 7.4 to obtain:

$$\begin{aligned} \alpha(\text{HIDE} \circ \text{COM} \circ \text{EXEC}^1)(\mathcal{A}) &\leq \\ \alpha(\text{HIDE}^0 \circ \text{COM} \circ \text{EXEC}^0, \text{HIDE}^1 \circ \text{COM} \circ \text{EXEC}^0)(\mathcal{A}) &+ \\ 2 \cdot \epsilon_{\text{SHVZK}} & \end{aligned}$$

From Lemma 7.6 we get  $\epsilon_{\text{SHVZK}} = 0$  which leaves us with

$$\begin{aligned} \alpha(\text{HIDE} \circ \text{COM} \circ \text{EXEC}^1)(\mathcal{A}) &\leq \\ \alpha(\text{HIDE}^0 \circ \text{COM} \circ \text{EXEC}^0, \text{HIDE}^1 \circ \text{COM} \circ \text{EXEC}^0)(\mathcal{A}) & \end{aligned}$$

To finish the proof we show that  $\text{HIDE}^0 \circ \text{COM} \circ \text{EXEC}^0$  and  $\text{HIDE}^1 \circ \text{COM} \circ \text{EXEC}^0$  are perfectly indistinguishable using Theorem 2.4. After inlining package composition and simplification of the if-statement we end up with the code comparison found in Table 6.

Because both sides use COM to store the value of  $e$  we cannot use equality of heaps as the invariant since the two sides store different values. Instead, we rely on the equality of heaps on all locations except the location of  $e$ . With the invariant fixed we now show equivalence between the two programs.

Looking at Table 6, we observe that the random sampling of  $e$  in the right-hand program has no counterpart on the left side. However, the particular choice of value for  $e$  on the right is not important:  $e$  simply gets used to compute  $z$  and gets stored in  $e\_loc$ . We can thus remove the sampling by appealing to the `sample-irrelevant` rule. This transformation is possible since our invariant allows us to ignore the value of  $e$  stored in memory on both sides. We are then left with the two sides being equal barring the computation of the value  $z$  and storing the value of  $z$ . Fortunately, the relational program logic supports asynchronous rules for writing memory. With this we can synchronize the two programs to the return statement. Last, we can conclude perfect equivalence since the return values are equal and no memory operations altered any locations except the locations ignored by the invariant.  $\square$



## 8 RELATED WORK

SSProve is the first verification framework for SSP, yet the formal verification of cryptographic proofs in other styles has been intensely investigated [8]. In this section we survey the closest related work in this space.

CertiCrypt [17] is a foundational Coq framework for game-based cryptographic proofs. CertiCrypt does not support modular proofs and is no longer maintained, yet it is seminal work that has inspired many other tools in this space, such as EasyCrypt, FCF, etc. The logic we introduce in §4 is also inspired by the probabilistic relational Hoare logic at the core of CertiCrypt.

FCF [51] is a more recent foundational Coq framework for cryptographic proofs that was used to verify the HMAC implementations in OpenSSL [20] and mbedTLS [66]. In contrast to CertiCrypt’s (and EasyCrypt’s) deep embedding of a probabilistic While language, FCF represents code with finite probabilities and non-termination using a monadic embedding, similar to the free monad we use for code in §3.1. The advantage of such an embedding is that code can be both easily manipulated as a syntactic object (e.g., to define package composition in §3.1) and easily lifted to a probability monad when needed (§3.2 and §5.2), all without leaving Gallina, the internal language of Coq. This monadic representation of computational effects also paves the way towards a more modular treatment of programs exhibiting effects of different nature such as communications with an external process. We are not aware of any formalization of SSP on top of FCF, although it seems possible in principle.

EasyCrypt [12, 14] is a proof assistant and verification tool specifically designed and built from scratch for game-based cryptographic proofs. This state-of-the-art tool has been used, for instance, to prove security for Amazon Web Services’ Key Management Service [4]. EasyCrypt’s good integration with automatic theorem provers (e.g., SMT solvers) is helpful for such large proofs, even if it does come at a cost in terms of trusted computing base.

EasyCrypt also comes with an ML-style module system [9]. EasyCrypt’s parameterized modules are, however, quite different from parameterized games in SSP (parameterized module instantiation in EasyCrypt has cloning semantics, i.e., each instance gets a separate copy of the module’s state). Moreover, EasyCrypt functors—which can to some extent be used to represent packages with imports—are not first class, so SSP-style laws cannot even be stated. While none of these is a showstopper, it leads to a quite different default style for writing modular proofs.

In very recent work, Dupressoir et al. [32] show that with enough workarounds they can code up in EasyCrypt the SSP proof of Brzuska et al. [25] for the Cryptobox [21] KEM-DEM [30], and discuss the strengths and shortcomings of EasyCrypt for formalizing SSP-style proofs. Our KEM-DEM example has similar complexity, but moreover we focus on providing a *general framework* for SSP proofs, including definitions of SSP packages, their composition, and the corresponding algebraic laws. SSProve also includes an `assert` operation, and a faithful representation of the SSP memory model, allowing to express SSP proofs more naturally.

EasyUC [29] aims to address the lack of composability in game-based proofs by formalizing the Universal Composability (UC) framework [27] using EasyCrypt. EasyUC replaces the interactive Turing machines in UC with EasyCrypt functions. It was used to prove a secure messaging protocol composed of Diffie-Hellman and one-time pad. More recent work develops a DSL [28] on top of EasyUC for hiding away the boilerplate needed to mediate between procedure-based communication in EasyCrypt and co-routine-based communication in the UC framework. Barbosa et al. [9] add automatic complexity analysis to EasyCrypt and use it for another formalization of UC. SSP was in part inspired by the UC framework, but focuses on making game-based proofs more modular and scalable, without targeting simulation-based security or universal composability. A more precise comparison between SSP and UC proofs would be interesting.

CryptHOL [18] is a foundational framework for game-based proofs that uses the theory of relational parametricity to achieve automation in the Isabelle/HOL proof assistant. It also makes use of the extensive mathematical libraries of Isabelle/HOL. More proof engineering and automation would be needed for SSProve to have a chance at matching CryptHOL’s formalization of ElGamal or PRF-based encryption. CryptHOL [42] has been also used to formalize Constructive Cryptography [46], another composable framework that inspired SSP, and the example of a one-time pad. While there is some similarity between their converters and SSP’s packages, to our knowledge a more precise comparison has not yet been undertaken.

ILC [41] is a process calculus modeling some of the key ideas behind the UC framework, in particular its co-routine based communication mechanism, while completely abstracting away from interactive Turing machines. Their work has not yet been formalized in a proof assistant.

IPDL [50] is another recent Coq framework for cryptographic proofs. Although their motivation is similar to SSP and their interaction sets are reminiscent of packages, the relation to other composable frameworks has not been worked out.

Packages have been motivated by ML modules [56]. Sequential composition is similar to the usual composition of modules. No specific theory for probabilistic programming languages with stateful modules seems to be available, but Sterling and Harper [61] provide a general module system. It would be interesting to specialize it to probabilistic stateful programs and compare it to packages.

## 9 FUTURE WORK

The high-level proofs done on paper in the miTLS project [22, 23, 34] were the main inspiration for the SSP methodology and it would be an interesting challenge to scale SSProve to large machine-checked proofs in the future. This would for a start require more work on proof engineering and automation. The problem of verifying such large proofs all the way down to low-level efficient executable code is even more challenging, also given the extreme scale of a complete implementation for a protocol like TLS. Achieving this in Coq would probably require integrating with projects such as VST [6] or FiatCrypto [33].

An alternative would be to port SSProve to  $F^*$  [63], where at least functional correctness can be verified at that scale. Still many challenges would remain, including extending  $F^*$  to probabilistic verification, internalizing  $F^*$  modules, and extending the SSP methodology to support type abstraction and procedures with specifications. In less ambitious recent work that is still unfinished, Kohbrok et al. [39] have implemented vanilla SSP packages in  $F^*$  and attempted to automate state-separating proofs based on a library for partial setoids.

We would also like to extend SSProve to extra side-effects such as non-termination and I/O and also to  $F^*$ -style sub-effecting [63]. The effect-modular semantic model from §5 should make this easier, and we hope to be able reuse the Interaction Trees framework [60, 65], and maybe also take inspiration from CryptHOL [18].

## ACKNOWLEDGMENTS

We are grateful to Arthur Azevedo de Amorim, Théo Laurent, and Ramkumar Ramachandra for their technical support and for participating in stimulating discussions. We are grateful to Markulf Kohlweiss for pointing us a bug in our modelling of public-key encryption schemes. Fortunately, the bug was easy to fix due to our flexible automation. We also thank all the other authors of [25] for their prompt fix of the informal proof of security for KEM-DEM. Finally we are grateful to the anonymous reviewers for their valuable feedback. This work was in part supported by the European Research Council under ERC Starting Grant SECOMP (715753), by AFOSR grant *Homotopy type theory and probabilistic computation* (12595060), by the Concordium Blockchain Research Center at

Aarhus University, by Nomadic Labs via a grant on the *Evolution, Semantics, and Engineering of the F\* Verification System*, and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) as part of the Excellence Strategy of the German Federal and State Governments – EXC 2092 CASA - 390781972. Antoine Van Muylder holds a PhD Fellowship from the Research Foundation – Flanders (FWO).

## REFERENCES

- [1] Carmine Abate, Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Cătălin Hrițcu, Kenji Maillard, and Bas Spitters. 2021. SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq. In *CSF. IEEE*. <https://doi.org/10.1109/CSF51468.2021.00048>
- [2] Reynald Affeldt, Cyril Cohen, Marie Kerjean, Assia Mahboubi, Damien Rouhling, and Kazuhiko Sakaguchi. 2020. Competing inheritance paths in dependent type theory: a case study in functional analysis. In *IJCAR 2020 - International Joint Conference on Automated Reasoning*. Paris, France, 1–19.
- [3] Reynald Affeldt, Cyril Cohen, Marie Kerjean, Assia Mahboubi, Damien Rouhling, Kazuhiko Sakaguchi, and Pierre-Yves Strub. 2021. mathcomp-analysis. Analysis library compatible with Mathematical Components. <https://github.com/math-comp/analysis>
- [4] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Matthew Campagna, Ernie Cohen, Benjamin Grégoire, Vitor Pereira, Bernardo Portela, Pierre-Yves Strub, and Serdar Tasiran. 2019. A Machine-Checked Proof of Security for AWS Key Management Service. In *CCS 2019*. ACM, 63–78. <https://doi.org/10.1145/3319535.3354228>
- [5] Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. 2015. Monads need not be endofunctors. *Logical Methods in Computer Science* 11, 1 (2015). [https://doi.org/10.2168/LMCS-11\(1:3\)2015](https://doi.org/10.2168/LMCS-11(1:3)2015)
- [6] Andrew W. Appel. 2011. Verified Software Toolchain - (Invited Talk). In *ESOP (Lecture Notes in Computer Science)*, Vol. 6602. Springer, 1–17. [https://doi.org/10.1007/978-3-642-19718-5\\_1](https://doi.org/10.1007/978-3-642-19718-5_1)
- [7] Philippe Audebaud and Christine Paulin-Mohring. 2006. Proofs of Randomized Algorithms in Coq. In *Mathematics of Program Construction*. Springer, 49–68.
- [8] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. 2021. SoK: Computer-aided cryptography. In *SP 2020-42nd IEEE Symposium on Security and Privacy*.
- [9] Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, and Pierre-Yves Strub. 2021. Mechanized Proofs of Adversarial Complexity and Application to Universal Composability. Cryptology ePrint Archive, Report 2021/156. <https://eprint.iacr.org/2021/156>
- [10] R. Barnes, B. Beurdouche, J. Millican, E. Omara, K. Cohn-Gordon, and R. Robert. 2020. The Messaging Layer Security (MLS) Protocol. IETF Draft. <https://tools.ietf.org/html/draft-ietf-mls-protocol-11>
- [11] Gilles Barthe, Juan Manuel Crespo, Benjamin Grégoire, César Kunz, Yassine Lakhnech, Benedikt Schmidt, and Santiago Zanella Béguelin. 2013. Fully automated analysis of padding-based encryption in the computational model. In *CCS'13*. ACM, 1247–1260. <https://doi.org/10.1145/2508859.2516663>
- [12] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. 2013. EasyCrypt: A Tutorial. In *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures (Lecture Notes in Computer Science)*, Vol. 8604. Springer, 146–166. [https://doi.org/10.1007/978-3-319-10082-1\\_6](https://doi.org/10.1007/978-3-319-10082-1_6)
- [13] Gilles Barthe, Thomas Espitau, Benjamin Grégoire, Justin Hsu, Léo Stefanescu, and Pierre-Yves Strub. 2015. Relational Reasoning via Probabilistic Coupling. In *LPAR-20*. 387–401. [https://doi.org/10.1007/978-3-662-48899-7\\_27](https://doi.org/10.1007/978-3-662-48899-7_27)
- [14] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. 2011. Computer-Aided Security Proofs for the Working Cryptographer. In *CRYPTO (Lecture Notes in Computer Science)*, Vol. 6841. Springer, 71–90. [https://doi.org/10.1007/978-3-642-22792-9\\_5](https://doi.org/10.1007/978-3-642-22792-9_5)
- [15] Gilles Barthe, Benjamin Grégoire, and Benedikt Schmidt. 2015. Automated Proofs of Pairing-Based Cryptography. In *CCS'15*. ACM, 1156–1168. <https://doi.org/10.1145/2810103.2813697>
- [16] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella-Béguelin. 2009. Formal certification of code-based cryptographic proofs. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*. 90–101. <https://doi.org/10.1145/1480881.1480894>
- [17] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. 2009. Formal certification of code-based cryptographic proofs. In *POPL*. 90–101.
- [18] David A. Basin, Andreas Lochbihler, and S. Reza Sefidgar. 2020. CryptHOL: Game-Based Proofs in Higher-Order Logic. *J. Cryptol.* 33, 2 (2020), 494–566. <https://doi.org/10.1007/s00145-019-09341-z>
- [19] Mihir Bellare and Phillip Rogaway. 2006. The Security of Triple Encryption and a Framework for Code-Based Game-Playing Proofs. In *Advances in Cryptology - EUROCRYPT 2006, 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques, St. Petersburg, Russia, May 28 - June 1, 2006, Proceedings (Lecture Notes in Computer Science)*, Serge Vaudenay (Ed.), Vol. 4004. Springer, 409–426. [https://doi.org/10.1007/11761679\\_25](https://doi.org/10.1007/11761679_25)
- [20] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. 2015. Verified Correctness and Security of OpenSSL HMAC. In *24th USENIX Security Symposium*. USENIX Association, 207–221. <https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/beringer>
- [21] Daniel J. Bernstein. 2009. Cryptography in NaCl. <https://cr.yp.to/highspeed/naclcrypto-20090310.pdf>
- [22] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Jianyang Pan, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella Béguelin, and Jean Karim Zinzindohoue. 2017. Implementing and Proving the TLS 1.3 Record Layer. *IEEE S&P* (2017).

- [23] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Santiago Zanella Béguelin. 2014. Proving the TLS Handshake Secure (As It Is). In *CRYPTO'14 (Lecture Notes in Computer Science)*, Vol. 8617. Springer, 235–255. [https://doi.org/10.1007/978-3-662-44381-1\\_14](https://doi.org/10.1007/978-3-662-44381-1_14)
- [24] Bruno Blanchet. 2006. A Computationally Sound Mechanized Prover for Security Protocols. In *IEEE S&P*. IEEE Computer Society, 140–154. <https://doi.org/10.1109/SP.2006.1>
- [25] Chris Brzuska, Antoine Delignat-Lavaud, Cédric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. 2018. State Separation for Code-Based Game-Playing Proofs. In *ASIACRYPT*. Springer International Publishing, Cham, 222–249. <https://eprint.iacr.org/2018/306>
- [26] D. Butler, A. Lochbihler, D. Aspinall, and A. Gascón. [n.d.]. Formalising  $\Sigma$ -Protocols and Commitment Schemes Using CryptHOL. 65, 4 ([n.d.]), 521–567. <https://doi.org/10.1007/s10817-020-09581-w>
- [27] Ran Canetti. 2020. Universally Composable Security. *J. ACM* 67, 5 (2020), 28:1–28:94. <https://doi.org/10.1145/3402457>
- [28] Ran Canetti, Assaf Kfoury, Alley Stoughton, Mayank Varia, Gollamudi Tarakaram, and Tomislav Petrovic. 2021. UC Domain Specific Language. unpublished. <https://github.com/easyuc/EasyUC/tree/master/uc-dsl>
- [29] Ran Canetti, Alley Stoughton, and Mayank Varia. 2019. EasyUC: Using EasyCrypt to Mechanize Proofs of Universally Composable Security. In *CSF*. IEEE, 167–183.
- [30] Ronald Cramer and Victor Shoup. 2003. Design and Analysis of Practical Public-Key Encryption Schemes Secure against Adaptive Chosen Ciphertext Attack. *SIAM J. Comput.* 33, 1 (2003), 167–226. <https://doi.org/10.1137/S0097539702403773>
- [31] Ivan Damgaard. [n.d.]. On Sigma-Protocols. lecture notes, Aarhus University. <https://cs.au.dk/~ivan/Sigma.pdf>
- [32] François Dupressoir, Konrad Kohbrok, and Sabine Oechsner. 2021. Bringing State-Separating Proofs to EasyCrypt - A Security Proof for Cryptobox. *Cryptology ePrint Archive*, Report 2021/326. <https://eprint.iacr.org/2021/326>
- [33] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala. 2019. Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises. In *IEEE S&P*. <https://doi.org/10.1109/SP.2019.00005>
- [34] 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*, Yan Chen, George Danezis, and Vitaly Shmatikov (Eds.). ACM, 341–350. <https://doi.org/10.1145/2046707.2046746>
- [35] Michèle Giry. 1982. A categorical approach to probability theory. In *Categorical Aspects of Topology and Analysis*, B. Banaschewski (Ed.). Springer, 68–85. [https://www.chrisstucchio.com/blog\\_media/2016/probability\\_the\\_monad/categorical\\_probability\\_giry.pdf](https://www.chrisstucchio.com/blog_media/2016/probability_the_monad/categorical_probability_giry.pdf)
- [36] Shai Halevi. 2005. A plausible approach to computer-aided cryptographic proofs. *IACR Cryptol. ePrint Arch.* (2005), 181. <http://eprint.iacr.org/2005/181>
- [37] Shin-ya Katsumata and Tetsuya Sato. 2013. Preorders on Monads and Coalgebraic Simulations. In *Foundations of Software Science and Computation Structures - 16th International Conference, FOSSACS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. 145–160. [https://doi.org/10.1007/978-3-642-37075-5\\_10](https://doi.org/10.1007/978-3-642-37075-5_10)
- [38] G M Kelly. [n.d.]. *Basic Concepts of Enriched Category Theory*. 143 pages. Reprint of the 1982 original [Cambridge Univ. Press, Cambridge; MR0651714].
- [39] Konrad Kohbrok, Markulf Kohlweiss, Tahina Ramananandro, and Nikhil Swamy. 2020. Relational F\* for State Separating Cryptographic Proofs. F\* wiki article. [https://github.com/FStarLang/FStar/wiki/Relational-F\\*-for-State-Separating-Cryptographic-Proofs](https://github.com/FStarLang/FStar/wiki/Relational-F*-for-State-Separating-Cryptographic-Proofs)
- [40] Sheng Liang, Paul Hudak, and Mark P. Jones. 1995. Monad Transformers and Modular Interpreters. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, Ron K. Cytron and Peter Lee (Eds.). ACM Press, 333–343. <https://doi.org/10.1145/199448.199528>
- [41] Kevin Liao, Matthew A. Hammer, and Andrew Miller. 2019. ILC: a calculus for composable, computational cryptography. In *PLDI*. ACM, 640–654.
- [42] Andreas Lochbihler, S. Reza Sefidgar, David A. Basin, and Ueli Maurer. 2019. Formalizing Constructive Cryptography using CryptHOL. In *CSF*. IEEE, 152–166. <https://doi.org/10.1109/CSF.2019.00018>
- [43] Saunders Mac Lane. 1978. *Categories for the Working Mathematician* (2 ed.). Graduate Texts in Mathematics, Vol. 5. Springer, New York, NY.
- [44] Assia Mahboubi and Enrico Tassi. 2021. Mathematical components. Online book. <https://math-comp.github.io/mcb/>
- [45] Kenji Maillard, Cătălin Hrițcu, Exequiel Rivas, and Antoine Van Muylder. 2020. The next 700 relational program logics. *Proc. ACM Program. Lang.* 4, POPL (2020), 4:1–4:33. <https://doi.org/10.1145/3371072>
- [46] Ueli Maurer. 2011. Constructive Cryptography - A New Paradigm for Security Definitions and Proofs. In *Theory of Security and Applications - Joint Workshop, TOSCA'11 (Lecture Notes in Computer Science)*, Vol. 6993. Springer, 33–56. [https://doi.org/10.1007/978-3-642-27375-9\\_3](https://doi.org/10.1007/978-3-642-27375-9_3)
- [47] Ueli Maurer and Renato Renner. 2011. Abstract Cryptography. In *Innovations in Computer Science - ICS'11*. Tsinghua University Press, 1–21. <http://conference.iis.tsinghua.edu.cn/ICS2011/content/papers/14.html>

- [48] Robin Milner, Joachim Parrow, and David Walker. 1992. A Calculus of Mobile Processes, I. *Inf. Comput.* 100, 1 (1992), 1–40. [https://doi.org/10.1016/0890-5401\(92\)90008-4](https://doi.org/10.1016/0890-5401(92)90008-4)
- [49] Eugenio Moggi. 1989. Computational Lambda-Calculus and Monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*. IEEE Computer Society, 14–23. <https://doi.org/10.1109/LICS.1989.39155>
- [50] 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, Report 2021/147. <https://eprint.iacr.org/2021/147>
- [51] Adam Petcher and Greg Morrisett. 2015. The Foundational Cryptography Framework. In *Principles of Security and Trust - 4th International Conference, POST 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings (Lecture Notes in Computer Science)*, Riccardo Focardi and Andrew C. Myers (Eds.), Vol. 9036. Springer, 53–72. [https://doi.org/10.1007/978-3-662-46666-7\\_4](https://doi.org/10.1007/978-3-662-46666-7_4)
- [52] Gordon D. Plotkin and John Power. 2003. Algebraic Operations and Generic Effects. *Applied Categorical Structures* 11, 1 (2003), 69–94. <https://doi.org/10.1023/A:1023064908962>
- [53] Gordon D. Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (Lecture Notes in Computer Science)*, Giuseppe Castagna (Ed.), Vol. 5502. Springer, 80–94. [https://doi.org/10.1007/978-3-642-00590-9\\_7](https://doi.org/10.1007/978-3-642-00590-9_7)
- [54] Christoph Rauch, Sergey Goncharov, and Lutz Schröder. 2016. Generic Hoare Logic for Order-Enriched Effects with Exceptions. In *Recent Trends in Algebraic Development Techniques - 23rd IFIP WG 1.3 International Workshop, WADT 2016, Gregynog, UK, September 21-24, 2016, Revised Selected Papers*. 208–222. [https://doi.org/10.1007/978-3-319-72044-9\\_14](https://doi.org/10.1007/978-3-319-72044-9_14)
- [55] Eric Rescorla. 2018. The Transport Layer Security (TLS) Protocol Version 1.3. IETF RFC 5246. <https://tools.ietf.org/html/rfc8446>
- [56] Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. 2014. F-ing modules. *J. Funct. Program.* 24, 5 (2014), 529–607. <https://doi.org/10.1017/S0956796814000264>
- [57] Mike Rosulek. 2021. The Joy of Cryptography. Online textbook. <http://web.engr.oregonstate.edu/~rosulekm/crypto/>
- [58] Claus Schnorr. 1991. Efficient signature generation by smart cards. *Journal of Cryptology* 4 (01 1991), 161–174. <https://doi.org/10.1007/BF00196725>
- [59] Victor Shoup. 2004. Sequences of games: a tool for taming complexity in security proofs. *IACR Cryptol. ePrint Arch.* (2004), 332. <http://eprint.iacr.org/2004/332>
- [60] Lucas Silver and Steve Zdancewic. 2021. Dijkstra monads forever: termination-sensitive specifications for interaction trees. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28.
- [61] Jonathan Sterling and Robert Harper. 2021. Logical relations as types: Proof-relevant parametricity for program modules. *Journal of the ACM (JACM)* 68, 6 (2021), 1–47.
- [62] Pierre-Yves Strub. [n.d.]. xhl. <https://github.com/strub/xhl>
- [63] Nikhil Swamy, Cătălin Hrițcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F\*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 256–270. <https://www.fstar-lang.org/papers/mumon/>
- [64] Philip Wadler. 1990. Comprehending Monads. In *LFP'90 (Nice, France)*. ACM, New York, NY, USA, 61–78. <https://doi.org/10.1145/91556.91592>
- [65] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL (2020), 51:1–51:32. <https://dl.acm.org/doi/10.1145/3371119>
- [66] Katherine Q. Ye, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W. Appel. 2017. Verified Correctness and Security of mbedTLS HMAC-DRBG. In *CCS'17*. ACM, 2007–2020. <https://doi.org/10.1145/3133956.3133974>