SSProve: A foundational framework for modular cryptographic proofs in Coq

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
CARMINE ABATE, 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, by using algebraic laws to exploit the modular structure of composed protocols. While 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 SSProve we use it to mechanize the simple security proofs of ElGamal and PRF-based encryption. We also validate the SSProve approach by conducting two more substantial case studies: First, we mechanize an SSP security proof of the KEM-DEM public key encryption scheme, which led to the discovery of an error in the original paper proof that has since been fixed. Second, we use SSProve to formally prove security of the sigma-protocol zero-knowledge construction and the associated construction of commitment schemes. We instantiate this proof to 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: computer-aided cryptography, game-based proofs, state-separating proofs, modular proofs, machine-checked proofs, probabilistic relational program logic, formal verification

1 INTRODUCTION

Cryptographic proofs can be challenging to make fully precise and to rigorously check. This has caused a "crisis of rigor" [26] in cryptography that Shoup [76], Bellare and Rogaway [26], Halevi [48], 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 [22, 66] and Isabelle/HOL [23]. It has also led to the development of specialized proof assistants [17, 31] and automated verification tools for cryptographic proofs [16, 21, 31]. There are two key ideas

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

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 [71] or the upcoming MLS [15]. To address this issue, Brzuska et al. [34] have recently introduced *state-separating proofs (SSP)*, a methodology for modular game-based proofs, inspired by the paper proofs in the miTLS project [29, 30, 45], by prior compositional cryptography frameworks [37, 59, 60], and by process algebras [61]. 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, and adversarial packages in particular 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 [74], which teaches cryptographic proofs in a style very similar to SSP.

While the SSP methodology is promising, and has for instance been recently used for proofs of the TLS 1.3 Key Schedule [33] and of the MLS draft standard [32], the lack of a complete formalization made SSP only usable for informal paper proofs, not for machine-checked ones. The SSP paper [34] 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 for instance the meaning of their `assert` operator is neither explained nor 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 [22, 48, 66, 82], or does not exactly match the SSP package abstraction [17, 55] (see §8 for a comparison with this related work).

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 [34]; (2) low-level details are formally proved in a probabilistic relational program logic [17, 22, 66]. Importantly, we show a formal connection between these two proof styles in Theorem 2.4.

SSProve 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 [67]. This gives us a principled way to define sequential composition of packages based on an algebraic effect handler [69] and to give machine-checked proofs of the SSP package laws [34], some of which were treated informally on paper. We formalize the state of SSP packages in terms of a shared global memory and make precise the minimal state-separation requirements, by only requiring disjoint state between adversaries and the games with which they are composed.

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. [58], 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 [58] 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 could also allow one to more easily add extra side-effects and F*-style sub-effecting [82] to SSProve in the future.

We formalize several security proofs, starting with the simple illustrative example of PRF-based encryption of Brzuska et al. [34], followed by a security proof for ElGamal public key encryption inspired by *The Joy of Cryptography* textbook [74, Chapter 15.3]. We then put SSProve to the test by formalizing two, more interesting case studies: First, we mechanize the security proof of the KEM-DEM public key encryption scheme of Cramer and Shoup [40], which Brzuska et al. [34] used to illustrate the main ideas of SSP. The proof extensively uses the package laws of SSP and showcases formal reasoning with invariants. Second, we give a new proof of security of Σ-protocols in SSP style, and show how any Σ-protocol can be used to construct a commitment scheme. We define a concrete example of a Σ-protocol following Schnorr [75] and prove concrete security bounds.

We have already started to reap the benefits of mechanizing SSP in a proof assistant: our mechanization of the KEM-DEM proof of Brzuska et al. [34] has led us to find—in conjunction with Brzuska et al.—an error in their originally published paper proof. Brzuska et al. have since proposed a revised version of their theorem and proof, which we have adapted and fully mechanized 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 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.

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 PRF-based and ElGamal 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 a 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 a first larger case study, formalizing security of the KEM-DEM public key encryption scheme of Cramer and Shoup [40], following the proof of Brzuska et al. [34]. In §7 we present a novel formalization of Σ-protocols in SSProve as a second case study. Finally, §8 discusses related work and §9 future directions.

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, one-sided memory accesses, and state invariants; (iii) significantly expanded the semantics section to be more self-contained and accessible, and to draw connections to related approaches (§5); (iv) added Sections 6 and 7 presenting two new case

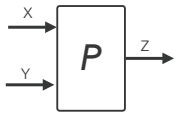


Fig. 1. Package P

```
package: P
mem:  n : nat
```

```
X(b):                Y():
  if b then            n ← n + 1
    return 1
  else
    return Z(n)
```

Fig. 2. Possible pseudocode implementation for P

studies; and (v) improved the related work section (§8). The paper has also gained a new author, and the author order has also slightly changed.

2 USING SSPROVE: KEY IDEAS AND EXAMPLES

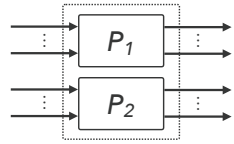
Formalizing the SSP methodology for high-level proofs allows us to formally link it to the methodology of probabilistic relational program logics for low-level proofs. In this section, we begin with a brief introduction to SSP (§2.1). Then, we present our new theorem connecting SSP to a probabilistic 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

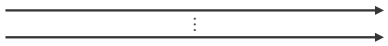
We begin by introducing our variant of the SSP methodology of Brzuska et al. [34]. The main concept behind this methodology is the *package*, which is a collection of procedure implementations that together manipulate a common piece of state, and that may depend on a set of external procedures. We refer 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 $\texttt{import}(P)$ to denote the set of procedure names the package P imports, and $\texttt{export}(P)$ to denote the names of the procedures it exports. The term *interface* is used to refer to such a set of procedure names.[1] 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 $P.\texttt{X}$.

In Figure 2, we can also see that the package implementation also depends on some memory location `n`, which can be read and updated as shown in procedures `X` and `Y` respectively. In SSP, such memory cells are implicitly initialized to default values depending on their type; here initially `n = 0`. In SSProve memory locations refer to a shared global memory and the declaration "`mem: n : nat`" in package P should be understood as: `n` is the only global memory location that is used by the procedures of P.

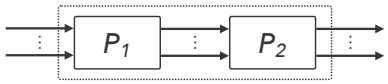
[1] In SSProve the procedure names within interfaces are also associated with argument and result *types*, but we omit this detail until §3.1.



(a) Parallel composition



(b) Identity package



(c) Sequential composition

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:

- *Sequential composition*: given two packages P_1 and P_2 with $\texttt{import}(P_1) \subseteq \texttt{export}(P_2)$, 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 P_1 and P_2 such that $\texttt{export}(P_1)$ and $\texttt{export}(P_2)$ are disjoint, then $P_1 \parallel P_2$ is the union of P_1 and P_2—it exports 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 $\texttt{ID}_I$, and we have that $\texttt{import}(\texttt{ID}_I) = \texttt{export}(\texttt{ID}_I) = I$.

In SSProve sequential and parallel composition are defined even in cases in which the composed packages use the same global memory locations, which allows state sharing.

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:

$$P_1 \circ (P_2 \circ P_3) \overset{\text{code}}{\equiv} (P_1 \circ P_2) \circ P_3 \tag{1}$$

Graphically these laws are obtained by simply forgetting about the dashed boxes (which represent parenthesizing) and by stretching arrows. In the SSP methodology, the $\overset{\text{code}}{\equiv}$ symbol stands for code equality between the packages: two packages are equal if the implementations of their procedures are equal to each other. As in SSProve code equality corresponds precisely to syntactic equality (including using the same global memory locations), we will write $P = Q$ instead of $P \overset{\text{code}}{\equiv} Q$ 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 G^{01} contains two games that export the same procedures, i.e., $G^{01} = (G^0, G^1)$ such that $\texttt{export}(G^0) = \texttt{export}(G^1)$ and $\texttt{import}(G^b) = \emptyset$ for $b = 0, 1$. A *distinguisher* for a game pair is a package $\mathcal{D}$ with

$\texttt{import}(\mathcal{D}) = \texttt{export}(G^0) = \texttt{export}(G^1)$ and $\texttt{export}(\mathcal{D}) = \{\texttt{Run}\}$, where 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^b exports a single procedure $\texttt{Run} : \texttt{unit} \to \texttt{bool}$ as above, we denote by $\Pr[\texttt{true} \leftarrow G]$ the probability that $G.\texttt{Run}$ returns the boolean value true 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 $G^{01} = (G^0, G^1)$ is

$$\alpha(G^{01})(\mathcal{D}) = \left|\Pr\left[\texttt{true} \leftarrow \mathcal{D} \circ G^0\right] - \Pr\left[\texttt{true} \leftarrow \mathcal{D} \circ G^1\right]\right|$$

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

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

□

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 $G^0, G^1, G^2, \ldots, G^n$ so that a bound for $\alpha(G^0, G^n)(\mathcal{D})$ can be given by

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

Lemma 2.3 (Reduction). *Let $G^{01} = (G^0, G^1)$ be a game pair and let P be an arbitrary package. Then, for every distinguisher $\mathcal{D}$, we have*

$$\alpha(P \circ G^0, P \circ G^1)(\mathcal{D}) = \alpha(G^{01})(\mathcal{D} \circ P).$$

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

$$\begin{aligned}
\alpha(P \circ G^0, P \circ G^1)(\mathcal{D}) &= \left|\Pr\left[\texttt{true} \leftarrow \mathcal{D} \circ \left(P \circ G^0\right)\right] - \Pr\left[\texttt{true} \leftarrow \mathcal{D} \circ \left(P \circ G^1\right)\right]\right| \\
&= \left|\Pr\left[\texttt{true} \leftarrow (\mathcal{D} \circ P) \circ G^0\right] - \Pr\left[\texttt{true} \leftarrow (\mathcal{D} \circ P) \circ G^1\right]\right| \\
&= \alpha(G^{01})\,(\mathcal{D} \circ P)
\end{aligned}$$

□

As its name indicates, Lemma 2.3 is used to reduce the advantage of the distinguisher over a composed game pair $(P \circ G^0, P \circ G^1)$ to the advantage over part of the game pair (G^{01}), for which we know a bound. We will use both these SSP lemmas in §2.3.

Shared state. As mentioned above, in SSProve we use a shared global memory and composed packages can share state. In particular our sequential and parallel composition are defined even in cases in which the composed packages use the same memory locations. This was easy to formalize in Coq, and allowed us to prove formally that the algebraic laws for package composition as well as the two lemmas above hold even when the involved packages share state.

This treatment of state in SSProve is quite different from the original SSP [34], in which composed packages have to always be "state separated" (i.e., have disjoint state). To make the state of packages disjoint they allow for α-renaming of state variables—which shows up for instance in their definition of code equality. Such informal α-renaming conventions are generally more difficult to formalize in a proof assistant [2, 12, 83]. Yet in the absence of α-renaming, requiring state disjointness everywhere would only increase the proof burden and the clutter in the proved security results (it would require the adversary's state to be disjoint from *all intermediate* games in the proof).

Adversaries. State separation is, however, still crucial for defining adversaries against game pairs. An *adversary* $\mathcal{A}$ for a game pair is a distinguisher whose memory footprint is disjoint from the footprint of each game in the pair. We define adversaries, the memory footprint of a package, and disjointness of footprints more formally in §3.3.

Perfect game indistinguishability. We say that the games G^0 and G^1 of a game pair G^{01} are *perfectly indistinguishable* when $\alpha(G^{01})(\mathcal{A}) = 0$ 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. [22] in CertiCrypt. Judgments of this logic are of the form

$$\vDash \ \{(m_0, m_1).\ \phi\} \ \ c_0 \ \sim \ c_1 \ \ \{(m_0', a_0), (m_1', a_1).\ \psi\}$$

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 ϕ, the final memories m_0', m_1' and results a_0, a_1 satisfy the postcondition ψ. When writing pre- and postconditions 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).[2] This notation is 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 $G^{01} = (G^0, G^1)$ be a game pair with export interface $\mathcal{E} = \mathtt{export}(G^b)$. Moreover, assume that ψ 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 $f : A \to B \in \mathcal{E}$, we have that for all $a \in A$,

$$\vDash \ \{\psi\} \ \ G^0.f(a) \ \sim \ G^1.f(a) \ \ \{(m_0', b_0), (m_1', b_1).\ b_0 = b_1 \wedge \psi(m_0', m_1')\}$$

[2] We will still use the λ notation for programs.

```
package: PRF⁰
mem:  k : option {0,1}^n
Eval(x):
  if k = ⊥ then
    k <$ uniform {0,1}^n
  return prf(k, x)
```

```
package: PRF¹
mem:  T : map [{0,1}^n -> {0,1}^n]
Eval(x):
  if T[x] = ⊥ then
    T[x] <$ uniform {0,1}^n
  return T[x]
```

Fig. 4. Packages PRF^0 and PRF^1

then we can conclude that $\alpha(G^{01})(\mathcal{A}) = 0$ *for any adversary* $\mathcal{A}$.

Intuitively, we ask that both procedures, when run on memories satisfying ψ, yield results drawn from the same distribution and memories still satisfying ψ. 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. This can be understood as a bisimulation argument between packages, where transitions between states come from procedure calls. We illustrate how this theorem is used in the examples from the next two subsections.

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. [34] 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 [34] 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. [34] 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 pseudo-random function, expressed as being *indistinguishable* from a package doing random sampling.

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

$$\mathtt{prf} : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$$

where $\{0,1\}^n$ represents the set of n-bit sequences. It is possible to formalize and quantify the security of PRF-based encryption as the probability for an adversary to distinguish it from a package that samples from a uniform distribution (*real vs random* paradigm [74]). Concretely, given the packages PRF^0 and PRF^1 as in Figure 4, where "`<$ uniform S`" represents uniform sampling from set `S`, the advantage of an adversary $\mathcal{A}$ against the game $\mathsf{PRF}^{01} = (\mathsf{PRF}^0, \mathsf{PRF}^1)$ is defined using Definition 2.1 as follows:

$$\alpha(\mathsf{PRF}^{01})(\mathcal{A}) = \left|\Pr\left[\mathsf{true} \leftarrow \mathcal{A} \circ \mathsf{PRF}^0\right] - \Pr\left[\mathsf{true} \leftarrow \mathcal{A} \circ \mathsf{PRF}^1\right]\right|$$

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

```
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

```
package: IND-CPA^0
mem:  k : option {0,1}^n
```

```
Enc(msg):
  if k = ⊥ then
    k <$ uniform {0,1}^n
  (r,c) ← enc(k, msg)
  return (r,c)
```

```
package: IND-CPA^1
mem:  k : option {0,1}^n
```

```
Enc(msg):
  if k = ⊥ then
    k <$ uniform {0,1}^n
  msg' <$ uniform {0,1}^n
  (r,c) ← enc(k, msg')
  return (r,c)
```

Fig. 6. Packages $\mathsf{IND\text{-}CPA}^0$ and $\mathsf{IND\text{-}CPA}^1$

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 $\mathsf{IND\text{-}CPA}^{01}$, and the packages involved are introduced in Figure 6. Notice that in procedure $\mathsf{IND\text{-}CPA}^1$.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 pair $\mathsf{IND\text{-}CPA}^{01}$ 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 $\mathcal{A}$ against $\mathsf{IND\text{-}CPA}^{01}$ is then $\alpha(\mathsf{IND\text{-}CPA}^{01})(\mathcal{A})$.

Brzuska et al. [34] use a sequence of *game-hops* to bound $\alpha(\mathsf{IND\text{-}CPA}^{01})$ in terms of (a linear function of) the advantage $\alpha(\mathsf{PRF}^{01})$. 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, and as a result we obtain a bound on the advantage of the game consisting of the initial and final game.

In this example, $\mathsf{IND\text{-}CPA}^b$ is shown equivalent to a variant, $\mathsf{MOD\text{-}CPA}^b$, that gets the pad through the PRF, i.e., with a call to Eval of the package PRF^0 or PRF^1 (see Figure 7). By repeatedly applying

```
package: MOD-CPA^0
mem:
Enc(msg):
  r <$ uniform {0,1}^n
  pad ← Eval(r)
  c ← msg xor pad
  return (r, c)
```

```
package: MOD-CPA^1
mem:
Enc(msg):
  msg' <$ uniform {0,1}^n
  r <$ uniform {0,1}^n
  pad ← Eval(r)
  c ← msg' xor pad
  return (r, c)
```

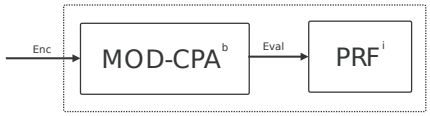


Fig. 7. Packages $\mathsf{MOD\text{-}CPA}^b$ import Eval from PRF^i

Lemma 2.2, we bound $\alpha(\mathsf{IND\text{-}CPA}^{01})(\mathcal{A})$ by

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

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

$$\alpha(\mathsf{PRF}^{01})(\mathcal{A} \circ \mathsf{MOD\text{-}CPA}^0) + \varepsilon_{stat.}(\mathcal{A}) + \alpha(\mathsf{PRF}^{01})(\mathcal{A} \circ \mathsf{MOD\text{-}CPA}^1).$$

where $\varepsilon_{stat.} = \alpha(\mathsf{MOD\text{-}CPA}^0 \circ \mathsf{PRF}^1,\ \mathsf{MOD\text{-}CPA}^1 \circ \mathsf{PRF}^1)$. The advantage of an attacker with respect to $\mathsf{MOD\text{-}CPA}^0$ and $\mathsf{MOD\text{-}CPA}^1$ is usually referred to as *statistical gap*, a polynomial function of the number of calls from the adversary (see [34, appendix A]). One could prove a precise bound based on the birthday paradox here [34, appendix A], but this would require formal reasoning about failure events [22]. It would be useful to extend SSProve in the future with a unary, union bounds logic for adding more precision to such steps [19].

We can informally interpret the formally proved bound above as saying that if the advantage of $\mathcal{A} \circ \mathsf{MOD\text{-}CPA}^b$ against PRF and the statistical gap are negligible then the advantage of $\mathcal{A}$ against $\mathsf{IND\text{-}CPA}^{01}$ is also negligible, under the additional assumption that $\mathcal{A}$ is probabilistic polynomial time and has disjoint state from PRF^1, so that $\mathcal{A} \circ \mathsf{MOD\text{-}CPA}^b$ is an adversary against PRF^{01}. While this disjointness assumption is needed at the top level, when applying Lemma 2.2 for reducing the advantage of the adversary $\mathcal{A}$ for game pair $\mathsf{IND\text{-}CPA}^{01}$, we are basically using $\mathcal{A}$ as a distinguisher, which does not introduce additional state disjointness requirements, i.e., for applying Lemma 2.2, the state of $\mathcal{A}$ is only required to be disjoint from $\mathsf{IND\text{-}CPA}^{01}$, not from $\mathsf{MOD\text{-}CPA}^b$ and PRF^b.

It remains to justify the two perfect indistinguishability statements above. These steps involve replacing an informal argument [34] by a fully formal one, moving to our probabilistic relational program logic. We will detail one of these steps: $\alpha(\mathsf{IND\text{-}CPA}^0,\ \mathsf{MOD\text{-}CPA}^0 \circ \mathsf{PRF}^0)(\mathcal{A}) = 0$. The other step, $\alpha(\mathsf{MOD\text{-}CPA}^1 \circ \mathsf{PRF}^0,\ \mathsf{IND\text{-}CPA}^1)(\mathcal{A}) = 0$, is analogous.

In order to prove this equivalence, Brzuska et al. [34] notice that the Enc procedures of $\mathsf{IND\text{-}CPA}^0$ and $\mathsf{MOD\text{-}CPA}^0 \circ \mathsf{PRF}^0$ (see Figure 8) return the same distributions of ciphertext when called on the

same msg. The two procedures are obtained by "inlining" the code of PRF^0.Eval inside $\mathsf{MOD\text{-}CPA}^0$, and by "unfolding" the code of enc.

$\mathsf{IND\text{-}CPA}^0$.Enc(msg)

```
if k = ⊥ then
  k <$ uniform {0,1}^n
r <$ uniform {0,1}^n
pad ← prf(k,r)
c ← msg xor pad
return (r, c)
```

$(\mathsf{MOD\text{-}CPA}^0 \circ \mathsf{PRF}^0)$.Enc(msg)

```
r <$ uniform {0,1}^n
if k = ⊥ then
  k <$ uniform {0,1}^n
pad ← prf(k,r)
c ← msg xor pad
return (r, c)
```

Fig. 8. Enc procedures expanded

The left- and right-hand side procedures in Figure 8 only differ when $\mathsf{k} = \bot$, 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. [34] conclude informally that independence allows to "swap" the two operations. We instead use Theorem 2.4 to formally reduce $\alpha(\mathsf{IND\text{-}CPA}^0, \mathsf{MOD\text{-}CPA}^0 \circ \mathsf{PRF}^0)(\mathcal{A}) = 0$ 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,

$$
\begin{array}{ll}
\vDash & \{(m_0, m_1).\ m_0 = m_1\} \\
& \mathsf{IND\text{-}CPA}^0.\mathsf{Enc}(\mathsf{msg}) \\
\sim & (\mathsf{MOD\text{-}CPA}^0 \circ \mathsf{PRF}^0).\mathsf{Enc}(\mathsf{msg}) \\
& \{(m_0', rc_0), (m_1', rc_1).\ m_0' = m_1' \wedge rc_0 = rc_1\}
\end{array}
$$

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, "$m_0' = m_1' \wedge rc_0 = rc_1$". We use the following instance of the swap rule from §4.1, to formally justify this swapping:[3]

$$
\frac{\vDash \{m_0 = m_1\}\ \mathtt{x \leftarrow\$\ uniform\{0,1\}^n} \sim \mathtt{y \leftarrow\$\ uniform\{0,1\}^n}\ \{m_0' = m_1' \wedge c_0 = c_1\} \quad \vDash \{m_0 = m_1\}\ \mathtt{y \leftarrow\$\ uniform\{0,1\}^n} \sim \mathtt{x \leftarrow\$\ uniform\{0,1\}^n}\ \{m_0' = m_1' \wedge c_0 = c_1\}}{\vDash \{m_0 = m_1\}\ \mathtt{x \leftarrow\$\ uniform\ \{0,1\}^n\ ;\ y \leftarrow\$\ uniform\ \{0,1\}^n} \sim \mathtt{y \leftarrow\$\ uniform\ \{0,1\}^n\ ;\ x \leftarrow\$\ uniform\ \{0,1\}^n}\ \{m_0' = m_1' \wedge c_0 = c_1\}
$$

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 [74, 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 key 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

[3]Here we omit quantifications in pre- and postconditions for conciseness.

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

```
Enc(pk, msg):
  rnd <$ uniform {0, ..., n-1}
  c_rnd ← g^rnd
  shs ← pk^rnd
  c_msg ← msg * shs
  return (c_rnd, c_msg)

Dec(sk, (c_rnd, c_msg)):
  return c_msg * (c_rnd^sk)^-1
```

Fig. 9. Algorithms for ElGamal encryption scheme

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, $\mathsf{pk} \in \mathcal{G}$. The key generation algorithm (KeyGen in Figure 9) generates a secret key that is a random number $\mathsf{sk} \in \{0, \ldots, n-1\}$ 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 $(\_)^{-1}$. Encryption works probabilistically, generating an ephemeral key rnd to derive a shared secret shs which is used to encrypt the plaintext message msg.

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 [74, Claim 15.5]. We leave the formalization of this last result as future work and discuss hereafter our proof of OT-CPA in SSProve.

```
package: DDH^0
mem:  pk : option pubKey
      sk : option secKey
```

```
Query()
  sk <$ uniform {0, ..., n-1}
  rnd <$ uniform {0, ..., n-1}
  pk ← g^sk
  return (g^sk, g^rnd, g^sk·rnd)
```

```
package: DDH^1
mem:  pk : option pubKey
      sk : option secKey
```

```
Query()
  sk <$ uniform {0, ..., n-1}
  rnd <$ uniform {0, ..., n-1}
  rnd' <$ uniform {0, ..., n-1}
  pk ← g^sk
  return (g^sk, g^rnd, g^rnd')
```

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

```
package: CPA^0
mem:  pk : option pubKey
      sk : option secKey
      counter : nat
```

```
Get_pk():
  return pk

Challenge(msg):
  assert counter = 0
  (pk, sk) ← KeyGen()
  (c_rnd, c_msg) ← Enc (pk, msg)
  counter++
  return (c_rnd, c_msg)
```

```
package: CPA^1
mem:  pk : option pubKey
      sk : option secKey
      counter : nat
```

```
Get_pk():
  return pk

Challenge(msg):
  assert counter = 0
  (pk, sk) ← KeyGen()
  (c_rnd, c_msg) <$ uniform G × G
  counter++
  return (c_rnd, c_msg)
```

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

The security property OT-CPA is expressed in terms of the advantage against game pair CPA^{01} in Figure 11. An adversary $\mathcal{A}$ can call `Get_pk()` and get the public key, if already initialized.[4] 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}) `<$ uniform` $\mathcal{G} \times \mathcal{G}$, i.e., a pair of group elements sampled from the uniform distribution on $\mathcal{G} \times \mathcal{G}$.

```
package: MOD-CPA
mem:  pk : option pubKey
      counter : nat
```

```
Get_pk():
  return pk

Challenge(msg):
  assert counter = 0
  (pk, c_rnd, shs) ← Query()
  counter++
  c_msg ← msg * shs
  return (c_rnd, c_msg)
```

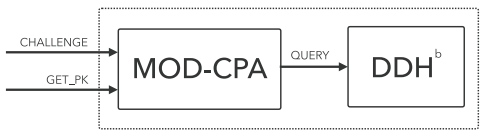


Fig. 12. Package `MOD-CPA` imports `Query` from DDH^b

[4] In a previous version of this work [1] we were – erroneously – not providing `Get_pk()` to the adversary, so the result was not a proper *public-key* scheme. We thank Markulf Kohlweiss for making us aware of this flaw, which was easy to fix. Formalizing the connection between OT-CPA and CPA [74, Claim 15.5] would have likely also exposed this flaw.

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

$$\alpha(\mathsf{CPA}^{01})(\mathcal{A}) \leq \alpha(\mathsf{DDH}^{01})(\mathcal{A} \circ \mathsf{MOD\text{-}CPA}).$$

We once again obtain this result by repeatedly applying Lemma 2.2 to bound $\alpha(\mathsf{CPA}^{01})(\mathcal{A})$ by

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

We prove that the first and last advantages are null by proving the packages perfectly indistinguishable, and the remaining advantage is equal to $\alpha(\mathsf{DDH}^{01})(\mathcal{A} \circ \mathsf{MOD\text{-}CPA})$ by simple application of Lemma 2.3. It now remains to show the equivalences below:

Step $\alpha(\mathsf{CPA}^0, \mathsf{MOD\text{-}CPA} \circ \mathsf{DDH}^0)(\mathcal{A}) = 0$*:* We apply Theorem 2.4 and reduce the goal to a relational judgment between CPA^0`.Challenge(msg)` and $(\mathsf{MOD\text{-}CPA} \circ \mathsf{DDH}^0)$`.Challenge(msg)` for a generic plaintext `msg`, and where the invariant ψ is equality of memories. Inlining the code of `Query` provided by DDH^0 inside MOD-CPA 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 $\alpha(\mathsf{MOD\text{-}CPA} \circ \mathsf{DDH}^1, \mathsf{CPA}^1)(\mathcal{A}) = 0$*:* 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 $\mathsf{MOD\text{-}CPA} \circ \mathsf{DDH}^1$ returns $\mathtt{msg} * \mathtt{g}^{\mathtt{rnd'}}$ for a random `rnd'`. To have equality of memories as invariant ψ, we show that in $\mathcal{G}$, multiplication by `g^(_)` acts like a one time pad, which is a standard result [22, 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). In §2.1 we took some license regarding notation in order to stay close to the presentation of Brzuska et al. [34]. The code examples in the remainder of the paper more 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 [63, 84]. 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 [66] 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:[5]

[5]This type of raw code comes equipped with an induction principle (basically structural induction on trees), 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.

```
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 : dom op → raw_code A).
```

Some more explanations about `raw_code` are in order. The type parameter `A` indicates the result of a computation of type `raw_code A`. 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 the first argument `p : opsig` specifies the name of the procedure, the type of its argument (`src p`), and its return type (`tgt p`). The second argument `x` is the input value of type `src p` passed to the called procedure. 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) global memory 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 `dom op` (the domain of the subdistribution) 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.

$$\texttt{get}\ \ell\ (\lambda\ \texttt{x}_\ell\ .\ \texttt{put}\ \ell\ (\texttt{x}_\ell + 1)\ (\textbf{\texttt{return}}\ \texttt{x}_\ell)) \tag{2}$$

$$\begin{aligned}&\texttt{sample}\ (\texttt{uniform}\ \{0,1\}^n)\\ &\qquad(\lambda\ \texttt{y}\ .\ \texttt{call}\ \texttt{Prf}\ (\texttt{y},\ 101010)\ (\lambda\ \texttt{z}\ .\ \textbf{\texttt{return}}\ \texttt{z}))\end{aligned} \tag{3}$$

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 bit-string 101010, and returns the result.

Code with locations and interface. Raw code is merely a representation of syntax. To record which imported procedures and global memory locations are used, we introduce a corresponding predicate. We consider the code with respect to set of *locations* $\mathcal{L}$ and to an *import interface* $\mathcal{I}$ which is a set of procedure signatures (`opsig`) consisting of a name, an input type and an output type. The predicate checks that all reads and writes performed in the code are made to locations in $\mathcal{L}$ and that all imported procedures belong to $\mathcal{I}$. Concretely, the code in (2) uses locations in $\{\ell : \texttt{nat}\}$ and has the empty import interface, while (3) uses the empty set of locations and the interface $\{\texttt{Prf} : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n\}$. The type $\texttt{code}_{\mathcal{L},\mathcal{I}}$ is then simply defined as raw code that verifies the predicate corresponding to locations $\mathcal{L}$ and import interface $\mathcal{I}$:

```
code_{L,I} A = { c : raw_code A | has_locs_and_imports c 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 regarding these locations and interfaces 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 ℓ, and its continuation is a Coq function "`(λ xℓ . put ...)`" of type "`type ℓ → raw_code nat`" that receives the value stored at ℓ as its parameter x_ℓ. The continuation in turn performs a `put` operation, storing the value $x_\ell + 1$ at memory location

ℓ, and returns the value x_ℓ. The code thus corresponds to the expression written as ℓ++ in common imperative languages.

Variables. As demonstrated in example (2), we draw a strict distinction between a location ℓ, which can be accessed and updated via `get` and `put`, and the value stored in memory at location ℓ. In (2), this value is available in the continuation of `get` ℓ (λx_ℓ . `put` ...) as x_ℓ. Formally speaking, x_ℓ is an immutable Coq variable, and in (2) the location ℓ itself is a Coq variable of type `Location`. This distinction is already present in SSP [34, Def. 2], where locations correspond to "state variables" and the ambient, mathematical notion of variable is referred to as "local variable".

Memory initialization. As mentioned in §2.1, memory locations are implicitly initialized to default values depending on their type (for instance in Figure 2 `n` is initialized to `0`). Yet we frequently want to clearly distinguish the case when a location was implicitly initialized to a default value from the case when it was explicitly initialized. For instance, in Figure 6 we want to distinguish the case in which the key `k` has not yet been generated from the case in which it has been. To achieve this we define the type of the `k` as **`option`** `key`, using the following standard definition for the option type:

```
Inductive option A := ⊥ | Some (a : A).
```

Memory locations of type **`option`** `key` are implicitly initialized to the dummy value ⊥, which is different from any generated key, which is tagged with the constructor `Some`.

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

```
Definition c : code nat :=
    sample (uniform bool) (λb. if b then return m₁ else return m₂)
Definition κ : nat → code nat :=  λm. put ℓ m (return 0)
```

We would like to use `c` as an argument to κ, but the types don't match: κ 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 κ when a returned value is encountered, and recursively pushing κ into any other continuations.

```
Fixpoint bind (c : code A) (κ : A → code B) : code B :=
  match c with
  | return a⇒ κ a
  | call p v κ'⇒ call p v  (λ x . bind (κ' x) κ)
  | get l κ'⇒ get l  (λ v . bind (κ' v) κ)
  | put l v κ'⇒ put l v  (bind κ' κ)
  | sample op κ'⇒ sample op  (λ a . bind (κ' a) κ)
  end
```

An easy structural induction over `code` allows us to prove that `bind` satisfies the expected monad laws: bind is associative (`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`. In the code below, the pattern matching happens over the natural number `N`, which in Coq is represented in unary format, so it is either `0` (zero) or `S n` (read successor of `n`, for some natural number `n`).

```
Fixpoint for_loop (N : nat) (c : nat → code unit) : code unit :=
  match N with
  | 0 ⇒ c 0
  | S n⇒ bind  (for_loop n 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 (λb. 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. In the future, we hope to extend SSProve to potentially non-terminating loops, since the semantic models of probabilistic programs usually support general fixpoints (as further discussed in §5.5).

Standard subdistributions. Probabilistic operations denoting a collection of subdistributions we may sample from are included in the type `Op`, which is a parameter of our language. 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 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` is Coq's 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. This provides a simple and clearly defined semantics for assertion failures. While this is not the only choice [25, 43, 72], this formalizes our understanding of the following informal convention from the paper introducing SSP [34]: "all our definitions and theorems apply only to code that never violates assertions."

We can see this use of `assert` in Figure 11 from §2.4. The packages CPA^b ensure that the `Challenge` procedure can be called only once by running `assert counter = 0`. If the assertion succeeds, we may assume that `counter = 0` holds in the rest of the procedure, until `counter` is incremented. We can demonstrate how distinguishers interact with procedures calling `assert` by computing $\Pr[\mathsf{true} \leftarrow \mathcal{D} \circ \mathsf{CPA}^b]$ (as used in Definition 2.1) for a distinguisher $\mathcal{D}$ that calls `Challenge` twice on some fixed message before always returning `true`. Even though the distinguisher still returns `true` in this scenario, `assert` will fail the second time the distinguisher calls `Challenge`, and thus the subdistribution represented by $(\mathcal{D} \circ \mathsf{CPA}^b)$.Run becomes the null subdistribution. It follows that $\Pr[\mathsf{true} \leftarrow \mathcal{D} \circ \mathsf{CPA}^b] = 0$.

The simple use of `assert` above 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 more complex use of assertions, which we call "dependent", occurs in the following example. Consider the situation where we have a memory location ℓ : `Location` holding a key which is initialized to $\bot$. Formally, this amounts to the information `type` ℓ = **`option`** `key`, which may be presented in the signature of a package as `mem:` ℓ : **`option`** `key`. We did not distinguish immutable variables and locations in §2, but we carry out a careful analysis of memory initialization in the KEM-DEM case study in §6. For instance, we will see in Figure 14 (§6, Page 36) an implementation of a `Get()` procedure that returns the key stored at location `k_loc`. This procedure defines a partial function of type `unit` → `key`, that fails to return a value if the memory location has not yet been explicitly initialized (i.e., it is $\bot$).

```
mem: k_loc : option key
```

```
Get():
  k ← get k_loc
  assert (k ≠ ⊥) as kSome
  return (getSome k kSome)
```

We first retrieve the potentially not explicitly initialized key from memory as `k`, of type `option key`. We then check via a dependent assert that `k` is not $\bot$, and record the asserted condition as `kSome` of type `k` ≠ $\bot$. 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 when reasoning in the relational program logic. Indeed the continuation of dependent assert, called `kont` in the declaration below, has type `b` = **`true`** → `code A`. 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. Operationally `assertD` is similar to the simple `assert` above, but the actual Coq definition in terms of `fail` and a conditional is more complicated because it uses dependent elimination to type-check the success branch. Since such details are not important here, we refer the interested reader to the Coq source and only show the type of this operator:

```
Definition assertD A b (kont : b = true → code A) : code A.
```

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 κ matching the return type of `p`. In §3.3 we show how an implementation gets substituted for this placeholder via sequential package composition.

Notation. The use of continuations is pervasive in monadic code, and to alleviate the presentation we introduce the following more familiar notation.

`x` ← c_1 ; c_2	:=	`bind` c_1 (λx.c_2)
`x` ← `p(a)` ; `c`	:=	`call p a` (λx.`c`)
`x` ← `get` ℓ ; `c`	:=	`get` ℓ (λx.`c`)
`put` ℓ := `v` ; `c`	:=	`put` ℓ `v c`
`x` <$ `D` ; `c`	:=	`sample D` (λx.`c`)
`assert e as H` ; `c`	:=	`assertD e` (λH.`c`)

In code listings we will frequently omit the ; separator when it would occur at the end of a line. We will also write c_1 ; c_2 for _ ← c_1 ; c_2 when the continuation c_2 does not use its argument.

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 ℓ if its domain matches the type of ℓ, i.e. `f : type` $\ell \to$ `raw_code A` for some type `A`.

To see the full definition in action, we restate the procedure `Eval(x)` from Figure 4 more formally.

```
Definition Eval (x : {0,1}^n) : raw_code {0,1}^n :=
  val_k_opt ← get k ;
  val_k ← (match val_k_opt with
           | ⊥⇒ y <$ uniform {0,1}^n ;
                   put k := Some y ;
                   return y
           | Some val_k'⇒ return val_k'
           end) ;
  val_prf ← Prf(val_k, x) ;
  return val_prf.
```

Here we 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 ← … ; …`').

3.2 Semantics of cryptographic code

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

```
Pr_code c : mem→ SD (A × mem)
```

This semantics is similar to that of CertiCrypt [22]. The type `SD A` denotes the collection of all subdistributions over type `A`. Generally speaking, a subdistribution is a function $d : A \to \mathbb{R}$ 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` [4, 57], a Coq library from which we use the foundations it gives to discrete probability theory. 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 `mem` $\to$ `SD(– × mem)`.

Using this subdistribution semantics, we can formalize the notation $\Pr[b \leftarrow G]$ 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 : SD bool`, the type of subdistributions for booleans, and we precisely define $\Pr[b \leftarrow G] = \mathsf{d}(b)$ 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, i.e., functions from some `A` to `raw_code B`. 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 uses only locations in $\mathcal{L}$ and imports only from $\mathcal{I}$, and each procedure name listed in $\mathcal{E}$ is implemented by a procedure in RP of the appropriate type. Consider for instance the package `MOD-CPA` in Figure 13. The memory

```
package: MOD-CPA
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 MOD-CPA (repeated)

used, mem(MOD-CPA),[6] consists of two locations, `pk : option pubKey` and `counter : nat`. The import interface import(MOD-CPA) contains a single procedure `Query : unit → 𝒢 × 𝒢 × 𝒢`. There are two procedures implemented by MOD-CPA, yielding an export interface export(MOD-CPA) containing `Get_pk : unit → option pubKey` and `Challenge : 𝒢 → option (𝒢 × 𝒢)`.

We define composition of packages, following the intuition of Brzuska et al. [34]. Given two raw packages P, Q we 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 procedure for which we search, we use a dummy value instead. If the exports of P match the imports of Q—i.e. $\mathsf{import}(Q) \subseteq \mathsf{export}(P)$—then no dummy value will be used. Concretely, during the traversal each `call p a κ` node is replaced by

```
bind (P.p a) (λ x . link_P (κ x))
```

where $\mathtt{link}_P$ stands for the recursive call of the function composing P with the remaining code. Experts will recognize this transformation as an algebraic effect handler [24], 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 $\mathsf{mem}(Q \circ P) = \mathsf{mem}(P) \cup \mathsf{mem}(Q)$, $\mathsf{import}(Q \circ P) = \mathsf{import}(P)$ and $\mathsf{export}(Q \circ P) = \mathsf{export}(Q)$.

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 their exports are disjoint, i.e. $\mathsf{export}(P) \cap \mathsf{export}(Q) = \emptyset$, then this overlap situation does not happen and we have $\mathsf{mem}(P \parallel Q) = \mathsf{mem}(P) \cup \mathsf{mem}(Q)$, $\mathsf{import}(P \parallel Q) = \mathsf{import}(P) \cup \mathsf{import}(Q)$ and $\mathsf{export}(P \parallel Q) = \mathsf{export}(P) \cup \mathsf{export}(Q)$.

Private state. When formalizing composition in SSProve we do not impose a priori restrictions on the disjointness of the state that the composed packages 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 $\mathcal{A}$ and the games in a pair G^{01}. 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 the end of §2.1)

[6] Here we write $\mathsf{mem}(P)$ to denote the memory footprint of package P. In the Coq formalization, we instead use a relation on a package and a set of memory locations stating that the packages only uses memory locations in the set but not necessarily all of them. This is a technical detail and in the remainder of the paper we will stick to the set notation.

and corresponding theorem statements (e.g., Theorem 2.4). Formally, an *adversary* $\mathcal{A}$ to game pair G^{01} is thus a distinguisher for G^{01} with the additional assumption that $\mathtt{mem}(\mathcal{A})$ is disjoint from both $\mathtt{mem}(G^0)$ and $\mathtt{mem}(G^1)$.

This extra state assumption sets apart adversaries from distinguishers: we do not require that the state of a distinguisher be disjoint from the game pair it tries to distinguish, and we still proved Lemma 2.2 and Lemma 2.3 in §2.1. The difference between distinguishers and adversaries manifests, for instance, in the definition of perfect indistinguishability from the end of §2.1, which quantifies only over adversaries. Distinguishers that are not adversaries can otherwise directly access the game state and trivially distinguish the games we want to consider indistinguishable.

In the setting of SSProve, in which memory locations are global and not up to α-renaming, this fine-grained state separation is helpful, since not only it minimizes the burden of formally proving disjointness, but it also reduces the clutter in the statement of the final results.

Automation of side conditions. In our Coq formalization, we provide automation for the checking of side conditions required to build packages and their compositions. This includes checking predicates such as `has_locs_and_imports` to ensure that the implementation of a package is consistent with the expected interface and memory footprint. There are limitations, however, to the scope of the automation; specifically the checking disjointness of location sets is currently quite basic. We plan to improve this situation in future work.

3.4 Package laws

We formally proved the algebraic laws obeyed by packages as stipulated by Brzuska et al. [34]. 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 that

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

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

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

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

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" [34].

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

The logic we use is a variant of pRHL, a probabilistic relational Hoare logic introduced by Barthe et al. [22]. The logic exposes relational judgments of the form

$$\vDash \ \{\!(m_0, m_1).\ \phi\}\!\}\ \ c_0\ \sim\ c_1\ \ \{\!(m_0', a_0), (m_1', a_1).\ \psi\}\!\},$$

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 $m_0 : \mathsf{mem}, m_1 : \mathsf{mem} \vdash \phi : \mathbb{P}$ 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 $m_0' : \mathsf{mem}, a_0 : A_0, m_1' : \mathsf{mem}, a_1 : A_1 \vdash \psi : \mathbb{P}$ is a predicate on the values returned by the executed code, which is parameterized by the variables m_0' and m_1' representing the final state of the memory (after execution) and by the final values a_0 and a_1. As mentioned before, we will sometimes omit the quantifications when they are clear from the context. We will also abuse notation and sometimes write e.g., $\psi(m_0', a_0)(m_1', a_1)$ for the substitution of ψ with the given memories and values. The code fragments appearing in a judgment are drawn from the free monad $\mathsf{code}_{\mathcal{L},I}$ of §3.1, and meet the further requirement that no oracle calls `call o x k` appear in them (exactly as in §3.2). The precondition ϕ is defined to be a relation between initial memories (for instance, $m_0 = m_1$). Similarly the postcondition ψ 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. [18]. In the remainder of this subsection we describe a selection of our rules. The presentation does not contain all the rules employed in practice by SSProve, nor does it provide a canonical presentation of these rules: some rules are overlapping hence there are multiple ways to prove the same relational judgment, but the actual derivation might be simpler with this redundancy. We return to the question of the organization of rules after this presentation.

$$\frac{c : \mathsf{code}_{\mathcal{L}}\ A}{\vDash\ \{\!\{m_0 = m_1\}\!\}\ \ c\ \sim\ c\ \ \{\!(m_0', a_0), (m_1', a_1).\ m_0' = m_1' \wedge a_0 = a_1\}\!\}}\ \texttt{reflexivity}$$

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

$$\frac{\begin{array}{c} c_0 : \mathsf{code}_{\mathcal{L}_0}\ A_0 \qquad c_1 : \mathsf{code}_{\mathcal{L}_1}\ A_1 \\ \kappa_0 : A_0 \to \mathsf{code}_{\mathcal{L}_0}\ B_0 \qquad \kappa_1 : A_1 \to \mathsf{code}_{\mathcal{L}_1}\ B_1 \\ \vDash\ \{\!\{\phi\}\!\}\ \ c_0\ \sim\ c_1\ \ \{\!\{\chi\}\!\} \\ \forall a_0\ a_1.\ \vDash\ \{\!(m_0, m_1).\ \chi(m_0, a_0)(m_1, a_1)\}\!\}\ \ \kappa_0(a_0)\ \sim\ \kappa_1(a_1)\ \ \{\!\{\psi\}\!\} \end{array}}{\vDash\ \{\!\{\phi\}\!\}\ \ a_0 \leftarrow c_0\,;\ \kappa_0(a_0)\ \sim\ a_1 \leftarrow c_1\,;\ \kappa_1(a_1)\ \ \{\!\{\psi\}\!\}}\ \texttt{seq}$$

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

$$\frac{\begin{array}{c} c_0 : \mathsf{code}_{\mathcal{L}}\ A_0 \qquad c_1 : \mathsf{code}_{\mathcal{L}}\ A_1 \\ \vDash \{I\}\ c_0 \sim c_1\ \{(m_0', a_0), (m_1', a_1).\ I(m_0', m_1') \wedge \psi(m_0', a_0)(m_1', a_1)\} \\ \vDash \{I\}\ c_1 \sim c_0\ \{(m_1', a_1), (m_0', a_0).\ I(m_0', m_1') \wedge \psi(m_0', a_0)(m_1', a_1)\} \end{array}}{\vDash \{I\}\ c_0\ ;\ c_1 \sim c_1\ ;\ c_0\ \{(m_0', a_0), (m_1', a_1).\ I(m_0', m_1') \wedge \psi(m_0', a_0)(m_1', a_1)\}}\ \texttt{swap}$$

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.

$$\frac{\begin{array}{c} c_0,\ c_0' : \mathsf{code}_{\mathcal{L}}\ A_0 \qquad c_1 : \mathsf{code}_{\mathcal{L}'}\ A_1 \\ \vDash \{\phi\}\ c_0 \sim c_1\ \{\psi\} \qquad \mathtt{Pr\_code}\ c_0 = \mathtt{Pr\_code}\ c_0' \end{array}}{\vDash \{\phi\}\ c_0' \sim c_1\ \{\psi\}}\ \texttt{eqDistrL}$$

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.

$$\frac{\begin{array}{c} c_0 : \mathsf{code}_{\mathcal{L}}\ A_0 \qquad c_1 : \mathsf{code}_{\mathcal{L}}\ A_1 \\ \vDash \{(m_0, m_1).\ \phi(m_0, m_1)\}\ c_0 \sim c_1\ \{(m_0', a_0), (m_1', a_1).\ \psi(m_0', a_0)(m_1', a_1)\} \end{array}}{\vDash \{(m_1, m_0).\ \phi(m_0, m_1)\}\ c_1 \sim c_0\ \{(m_1', a_1), (m_0', a_0).\ \psi(m_0', a_0)(m_1', a_1)\}}\ \texttt{symmetry}$$

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

$$\frac{\begin{array}{c} c_0,\ c_1 : \mathbb{N} \to \mathsf{code}_{\mathcal{L}}\ \mathsf{unit} \qquad N : \mathbb{N} \\ \forall i.\ \vDash \{I\ i\}\ c_0\ i \sim c_1\ i\ \{I\ (i+1)\} \end{array}}{\vDash \{I\ 0\}\ \mathtt{for\_loop}\ N\ c_0 \sim \mathtt{for\_loop}\ N\ c_1\ \{I\ (N+1)\}}\ \texttt{for-loop}$$

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.

$$\frac{\begin{array}{c} c_0,\ c_1 : \mathsf{code}_{\mathcal{L}}\ \mathsf{bool} \qquad N : \mathbb{N} \\ \vDash \{I\ (\mathtt{true}, \mathtt{true})\}\ c_0 \sim c_1\ \{(m_0', b_0), (m_1', b_1).\ b_0 = b_1 \wedge I\ (b_0, b_1)(m_0', m_1')\} \end{array}}{\begin{array}{ll} \vDash & \{I\ (\mathtt{true}, \mathtt{true})\} \\ & \mathtt{do\_while}\ N\ c_0 \\ \sim & \mathtt{do\_while}\ N\ c_1 \\ & \{(m_0', b_0), (m_1', b_1).\ b_0 = b_1 = \mathtt{false} \vee I\ (\mathtt{false}, \mathtt{false})(m_0', m_1')\} \end{array}}\ \texttt{do-while}$$

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. [58]. We believe that, with some additional work, their ideas could be used to also support unbounded loops in SSProve (see §5.5 for details).

$$\frac{|A|, |B| < \omega \qquad f : A \to B\ \textit{bijective} \qquad \forall x.\ \vDash \{\phi\}\ c_0\ x \sim c_1\ (f\ x)\ \{\psi\}}{\vDash \{\phi\}\ a \leftarrow\$\ \mathtt{uniform}\ A\ ;\ c_0\ a \sim b \leftarrow\$\ \mathtt{uniform}\ B\ ;\ c_1\ b\ \{\psi\}}\ \texttt{uniform}$$

The `uniform` rule relates sampling from uniform distributions on finite sets A and B that are in a bijective correspondence. Note how it applies the bijection f in the continuation on the right-hand side.

$$\frac{\begin{array}{c} D : \mathsf{Op} \qquad \sum_{x \in |D|} D(x) = 1 \\ \vDash \ \{\phi\}\ c_0 \ \sim \ c_1\ \{\psi\} \qquad y \notin \mathsf{freevar}(c_0) \end{array}}{\vDash \ \{\phi\}\ y \mathrel{<\$} D\,;\ c_0 \ \sim \ c_1\ \{\psi\}}\ \texttt{dead-sample}$$

The code $y \mathrel{<\$} D\,;\ c_0$ 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.

$$\frac{\begin{array}{c} D : \mathsf{Op} \qquad \sum_{x \in |D|} D(x) = 1 \\ \forall y.\ \vDash \ \{\phi\}\ c_0\ y \ \sim \ c_1\ \{\psi\} \end{array}}{\vDash \ \{\phi\}\ y \mathrel{<\$} D\,;\ c_0\ y \ \sim \ c_1\ \{\psi\}}\ \texttt{sample-irrelevant}$$

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.

$$\frac{b_0,\ b_1 : \mathsf{bool}}{\vDash \ \{(m_0, m_1).\ b_0 = b_1\}\ \mathtt{assert}\ b_0 \ \sim \ \mathtt{assert}\ b_1\ \{(m_0', a_0), (m_1', a_1).\ b_0 = \mathtt{true} \wedge b_1 = \mathtt{true}\}}\ \texttt{assert}$$

The `assert` rule relates two `assert` commands, as long as "$b_0 = b_1$" holds before the commands. Note that while the precondition is a predicate on initial memories, nothing prevents it form talking about other things such as the booleans b_0 and b_1 quantified at the meta-level. It guarantees "$b_0 = \mathtt{true} \wedge b_1 = \mathtt{true}$" afterwards, ignoring the values a_0 and a_1 of type `unit`.

$$\frac{b : \mathsf{bool}}{\vDash \ \{b = \mathtt{true}\}\ \mathtt{assert}\ b \ \sim \ \mathtt{return}\ ()\ \{b = \mathtt{true}\}}\ \texttt{assertL}$$

The one-sided `assertL` rule specifies the behavior an `assert` with a `true` boolean, by relating it with `return ()`. Note that if a code fragment c_0 is shown to be related to an assertion failure $\vDash \ \{\mathrm{True}\}\ c_0 \ \sim \ \mathtt{assert\ false}\ \{\psi\}$, then c_0 must necessarily contain an assertion failure as well, i.e., correspond to the null sub-distribution. Indeed the (sound) model of our program logic, explained in §5, gives rise to a total correctness semantics [58] for assertion failures: assertion failures only relate to other assertion failures.

$$\frac{\begin{array}{c} b_0 : \mathsf{bool} \qquad \kappa_0 : b_0 = \mathtt{true} \to \mathsf{code}\ A_0 \\ b_1 : \mathsf{bool} \qquad \kappa_1 : b_1 = \mathtt{true} \to \mathsf{code}\ A_1 \\ pre \implies b_0 = b_1 \\ H_0 : b_0 = \mathtt{true},\ H_1 : b_1 = \mathtt{true} \vDash \ \{\phi\}\ \kappa_0\ H_0 \ \sim \ \kappa_1\ H_1\ \{\psi\} \end{array}}{\vDash \ \{\phi\}\ \mathtt{assert}\ b_0\ \mathtt{as}\ h_0\,;\ \kappa_0\ h_0 \ \sim \ \mathtt{assert}\ b_1\ \mathtt{as}\ h_1\,;\ \kappa_1\ h_1\ \{\psi\}}\ \texttt{assertD}$$

The `assertD` rule allows reasoning about the dependent version of `assert` where the continuation κ_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 ϕ has to ensure that b_0 and b_1 are either both true or both false. The continuations κ_i are defined only in case the assertions succeed. Under this assumption, here represented as the hypotheses H_0 and H_1, the continuations κ_i must be related for the same *pre* and *post* as the composite statements "$\texttt{assert}\ b_i\ \texttt{as}\ h_i\,;\ \kappa_i\ h_i$". The intuition for the validity of this rule is the following: if b_i is true, $\texttt{assert}\ b_i\ \texttt{as}\ h_i$ 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.

$$\frac{\ell : \mathcal{L} \qquad r : \mathsf{type}\ \ell \to \mathsf{code}_{\mathcal{L}}\ A \qquad v : \mathsf{type}\ \ell}{\vDash \quad \{m_0 = m_1\} \quad \begin{array}{l}\mathsf{put}\ \ell\ v\,;\ x \leftarrow \mathsf{get}\ \ell\,;\ r(x) \ \sim \\ \mathsf{put}\ \ell\ v\,;\ r(v)\ \ \{(m_0', a_0), (m_1', a_1).\ m_0' = m_1' \wedge a_0 = a_1\}\end{array}}\ \texttt{put-get}$$

The `put-get` rule states that looking up the value at location ℓ after storing v at ℓ 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 one-sided rules for `get` and `put` which update the pre- or postcondition accordingly.

$$\frac{\begin{array}{c}\ell : \mathcal{L}_0 \qquad \kappa : \mathsf{type}\ \ell \to \mathsf{code}_{\mathcal{L}_0}\ A_0 \qquad c : \mathsf{code}_{\mathcal{L}_1}\ A_1 \\ \forall x.\ \vDash \{(m_0, m_1).\ \phi(m_0, m_1) \wedge m_0[\ell] = x\}\ \kappa(x)\ \sim\ c\ \{\psi\}\end{array}}{\vDash \{\phi\}\ x \leftarrow \mathsf{get}\ \ell\,;\ \kappa(x)\ \sim\ c\ \{\psi\}}\ \texttt{get-lhs}$$

$$\frac{\begin{array}{c}\ell : \mathcal{L}_0 \qquad \kappa : \mathsf{type}\ \ell \to \mathsf{code}_{\mathcal{L}_0}\ A_0 \qquad v : \mathsf{type}\ \ell \qquad c : \mathsf{code}_{\mathcal{L}_1}\ A_1 \\ \forall m_0, m_1.\quad \phi(m_0, m_1) \Longrightarrow m_0[\ell] = v \qquad \vDash \{\phi\}\ \kappa(v)\ \sim\ c\ \{\psi\}\end{array}}{\vDash \{\phi\}\ x \leftarrow \mathsf{get}\ \ell\,;\ \kappa(x)\ \sim\ c\ \{\psi\}}\ \texttt{get-lhs-rem}$$

With `get-lhs`, the left-hand side program is able to read from a memory location while we record that information in the precondition. Dually, `get-lhs-rem` will recover that information from the precondition. We also use the information in the preconditions when dealing with memory invariants such as the one presented in Appendix A for the security proof of KEM-DEM.

The situation is slightly more complicated for one-sided writes because writing might break a postcondition. Typically, writing on only one side when the postcondition ensures that both memory locations are equal would (maybe temporarily) break said postcondition.

$$\frac{\begin{array}{c}\ell : \mathcal{L}_0 \qquad v : \mathsf{type}\ \ell \qquad c_0 : \mathsf{code}_{\mathcal{L}_0}\ A_0 \qquad c_1 : \mathsf{code}_{\mathcal{L}_1}\ A_1 \\ \vDash \{(m_0, m_1).\ \exists m.\ \phi(m, m_1) \wedge m_0 = m[\ell \mapsto v]\}\ c_0\ \sim\ c_1\ \{\psi\}\end{array}}{\vDash \{\phi\}\ \mathsf{put}\ \ell\ v\,;\ c_0\ \sim\ c_1\ \{\psi\}}\ \texttt{put-lhs}$$

$$\frac{\begin{array}{c}\ell : \mathcal{L}_0 \qquad v : \mathsf{type}\ \ell \qquad c_0 : \mathsf{code}_{\mathcal{L}_0}\ A_0 \qquad c_1 : \mathsf{code}_{\mathcal{L}_1}\ A_1 \\ \forall\ m_0\ m_1.\ \phi(m_0, m_1) \Longrightarrow \phi(m_0[\ell \mapsto v], m_1) \qquad \vDash \{\phi\}\ c_0\ \sim\ c_1\ \{\psi\}\end{array}}{\vDash \{(m_0, m_1).\ \exists m.\ \phi(m, m_1) \wedge m_0 = m[\ell \mapsto v]\}\ c_0\ \sim\ c_1\ \{\psi\}}\ \texttt{restore-pre-lhs}$$

Instead `put-lhs` modifies the precondition to state that the precondition ϕ was satisfied by a previous memory state, and that the current memory state is the same except that ℓ now points to v. Typically, when ϕ is an invariant—such as one stating the equality of the two memories—we relax the precondition temporarily until we reach a new state where it holds again, for instance by having a similar write on the right-hand side. Rule `restore-pre-lhs` is such an example—although much simplified—of how one can recover the precondition after a write, provided they can prove that the

precondition holds after the corresponding update of memory. 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, until one can show that the invariant is preserved by all these memory updates.

More generally, we define handy tactics to apply these rules immediately, as well as performing the necessary massaging of goals so that they become applicable. As such we have automation for swapping multiple lines at once and checking that the swap was legal. Moreover, these tactics rely on the hints mechanism of Coq and can thus be extended by the user.

Organization of the relational program logics rules. The rules of the relational program logic are not canonically derived but a few guidelines have been used to come up with them. These guidelines follow three independent criteria: the algebraic criterion, the historical criterion, and the practical criterion. The algebraic criterion follows the idea that the effects employed in the programming language can be modeled using algebraic structures, e.g., monads, and rules corresponding to the standard combinators of this algebraic presentation can be naturally expressed [58]. In particular, equationally presented monads such as the state monad, the exception monad or the Giry probability monad [47, 68] naturally induce reasoning rules corresponding to their equational theory [46]. The algebraic approach, however, falls short of providing a complete solution accounting for the distinction between one-sided and two-sided rules specific to a particular effect—e.g., rules for get and set when considering the state monad. For these rules, the historical presentation follows earlier work on (x)pRHL [18, 22], providing both one-sided and two-sided rules. Finally, the pragmatics of proving relational properties of programs in SSProve pushed for specific presentations of the rules, well tailored to streamlined applications in practice, in particular when considering the specifics of the Coq hosting environment, such as tactic language and incremental proof derivation through interactive use of existential variables and subgoals generation. The redundancy between rules created by these different approaches to relational program logic rules' design is not an issue in practice, since each of these rules is proved against the semantic model presented in §5 rather than assumed axiomatically. The question of completeness of the rules is somehow side-stepped in our setting by having an escape hatch using the relational semantics: in any case, if the rules we provide are not suitable to prove a particular judgment, one can always fall back to the underlying semantic model and prove an additional valid rule at that level. Ultimately, we validate the design of the rules present in SSProve via case studies, ensuring that the chose set of rules are indeed enough to obtain concrete interesting results.

4.2 Proof sketch for Theorem 2.4

If we denote by mem the type of memories, then a binary memory predicate

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

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

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

In other words, on locations outside of $\mathcal{L}_0$ and $\mathcal{L}_1$, ψ must ensure equality of corresponding values and nothing else.

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 ψ 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*

$$\vDash \ \{\psi\} \ c_0 \ \sim \ c_1 \ \{\phi\},$$

ψ holds on the initial memories, and for all m_0', m_1', x and y we have that

$$\phi(m_0', x)(m_1', y) \implies (x = a \iff y = b),$$

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 $\alpha(G^{01})(\mathcal{A}) = 0$, i.e.,

$$\left|\Pr\left[\mathsf{true} \leftarrow \mathcal{A} \circ G^0\right] - \Pr\left[\mathsf{true} \leftarrow \mathcal{A} \circ G^1\right]\right| = 0.$$

Using the hypothesis and the fact that the predicate ψ is a stable invariant, i.e., stable on sets of locations $\mathtt{mem}(G^0)$ and $\mathtt{mem}(G^1)$, we perform an induction on the code of the procedure $\mathcal{A}.\mathsf{Run}$, to establish

$$\vDash \ \{\psi\} \ \ (\mathcal{A} \circ G^0).\mathsf{Run}() \ \sim \ (\mathcal{A} \circ G^1).\mathsf{Run}() \ \ \{(m_0', b_0), (m_1', b_1).\ b_0 = b_1 \wedge \psi(m_0', b_0)(m_1', b_1)\}.$$

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:

$$\vDash \ \{\psi\} \ \texttt{get l (}\lambda\texttt{x.}\textbf{\texttt{return}}\ \texttt{x)} \ \sim \ \texttt{get l (}\lambda\texttt{x.}\textbf{\texttt{return}}\ \texttt{x)} \ \{(m_0', v_0), (m_1', v_1).\ v_0 = v_1 \wedge \psi(m_0', v_0)(m_1', v_1)\}$$

As ψ 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 (remember we are performing induction on the adversary's code), and we explicitly asked for the adversary memory $\mathtt{mem}(\mathcal{A})$ to be disjoint from $\mathtt{mem}(G^0)$ and $\mathtt{mem}(G^1)$. As the memory was not changed, the invariant ψ still holds on the final memory.

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

$$\Pr\left[\mathsf{true} \leftarrow \mathcal{A} \circ G^0\right] = \Pr\left[\mathsf{true} \leftarrow \mathcal{A} \circ G^1\right],$$

and therefore the advantage is 0. □

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 [58], instantiating it with probabilities, simple failures, and global state. We first give in §5.1 an overview of the framework of Maillard et al. [58]. 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 extensions that we

need in this work. Finally, in §5.5 we compare this methodology to other approaches for modelling relational program logics.

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 $\vDash\ c_0 \sim c_1 \{ w \}$ 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. [58] stems from the realization that not only effectful code can be modelled using monads, but specifications can too, and we can build semantics for $r\mathcal{L}$ using a so-called *relational effect observation* in three 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* $(A_0, A_1) \mapsto W(A_0, A_1)$ where A_0 corresponds to the return type of the left program, and A_1 the return type of the right program. The set $W(A_0, A_1)$ should be ordered by entailment of specifications, written $w \leq w'$.
(3) Finally, find an appropriate *relational effect observation* $\theta^{A_0 A_1} : M_0 A_0 \times M_1 A_1 \to W(A_0, A_1)$ mapping a pair of monadic 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 θ is specified we define a semantic judgment for $r\mathcal{L}$ as follows:

$$\vDash_\theta c_0 \sim c_1 \{ w \} \qquad \Longleftrightarrow \qquad \theta^{A_0 A_1}(c_0, c_1) \leq w$$

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 $\mathrm{BP}(A_0, A_1) := (A_0 \times A_1 \to \mathbb{P}) \to \mathbb{P}$, where $\mathbb{P}$ is the type of propositions. Intuitively a backward predicate transformer $w : \mathrm{BP}(A_0, A_1)$ maps a relational postcondition ϕ to a precondition $w\ \phi$ *sufficient* to ensure ϕ on the result of the executions of code fragments c_0, c_1 respecting w (i.e. for which $\vDash_\theta c_0 \sim c_1 \{ w \}$ for some θ). The preorder on $\mathrm{BP}(A_0, A_1)$ is given by reverse pointwise implication. For two backward predicate transformers $w_1, w_2 : \mathrm{BP}(A_0, A_1)$, 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 $\mathrm{toBP}(pre, post)$:

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

Note that BP does not form a monad: it takes two types A_0, A_1 as input but only returns one $(A_0 \times A_1 \to \mathbb{P}) \to \mathbb{P}$. 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, as discussed in [58] and explained here 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 $\mathtt{code}_{\mathcal{L},I}$ 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 [11, 47], following the track of §3.2, but ignoring considerations around state. This semantics assignment can in fact be seen as a monad morphism $\delta : \mathrm{F}_{Pr} \to \mathrm{SD}$.

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 θ_{Pr} is based on the notion of probabilistic coupling. A coupling $d : \texttt{coupling}(d_0, d_1)$ of two subdistributions $d_0 : \mathrm{SD}(A_0)$ and $d_1 : \mathrm{SD}(A_1)$ 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 $d_i : \mathrm{SD}(A_i)$ two subdistributions we define $\theta'_{Pr} : \mathrm{SD} \times \mathrm{SD} \to \mathrm{BP}$ by:

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

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

$$\theta_{Pr} \coloneqq \theta'_{Pr} \circ \delta^2 \quad : \mathrm{F}_{Pr} \times \mathrm{F}_{Pr} \to \mathrm{BP}.$$

Intuitively, if $w : BP(A_0, A_1)$ is obtained out of a $(pre, post)$ pair, the semantic judgment $\vDash_{\theta_{Pr}} c_0 \sim c_1 \{ w \}$ 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 $\vDash_{\theta_{Pr}} c_0 \sim c_1 \{ w \}$ 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 <$ uniform A`) are validated as well. Third, failing assertions at type A can be modelled 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 $d : \texttt{coupling}(s, s)$ of s with itself. Fifth, any relational effect observation θ validates a rule like `seq`. Such a rule is essentially a syntactic formulation of the fact that θ should preserve the monadic composition, which is true by definition.

The implementation of the relational effect observation θ_{Pr} in Coq depends 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 [3, 4], 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* [53]. A monad transformer maps monads M to monads $\mathrm{T}\,M$ and monad morphisms θ to monad morphisms $\mathrm{T}\,\theta$. 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 $\mathrm{StT}\,M(A) = S \to M(A \times S)$ with additional ability to read and write elements of S. Besides, a monad transformer comes equipped with a family of liftings $\texttt{lift}^{\mathrm{T}} : \forall M.\ M \to \mathrm{T}\,M$ coercing any computation in the original monad M to a computation in the extended effectful environment $\mathrm{T}\,M$. We generalize this to specification monads and build modularly an effect observation $\theta_{Pr,St}$ on top of θ_{Pr}:

$$\theta'_{Pr,St} \coloneqq \mathrm{StT}\,\theta_{Pr} \quad : \mathrm{StT}(\mathrm{F}^2_{Pr})(A_0, A_1) \to \mathrm{StT}(\mathrm{BP})(A_0, A_1)$$

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

$$\mathrm{StT}(\mathrm{F}^2_{Pr})(A_0, A_1) \coloneqq S_0 \times S_1 \to \mathrm{F}_{Pr}(A_0 \times S_0) \times \mathrm{F}_{Pr}(A_1 \times S_1),$$
$$\mathrm{StT}(\mathrm{BP})(A_0, A_1) \coloneqq (A_0 \times S_0 \times A_1 \times S_1 \to \mathbb{P}) \to S_0 \times S_1 \to \mathbb{P}.$$

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

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

Using the liftings $\mathtt{lift}^{\mathrm{StT}}$ provided by StT, we can build from any purely probabilistic relational judgment $\vDash_{\theta_{Pr}} c_0 \sim c_1 \ \{ \ w \ \}$, a relational judgment $\vDash_{\theta_{Pr,St}} c_0 \sim c_1 \ \{ \ \mathtt{lift}^{\mathrm{StT}} \ w \ \}$ in the state-aware model. This correspondence can be shown to form an embedding of logics: for every c_0, c_1, w free from state manipulation, derivations of $\vDash_{\theta_{Pr,St}} c_0 \sim c_1 \ \{ \ \mathtt{lift}^{\mathrm{StT}} \ w \ \}$ are in bijective correspondence with derivations of $\vDash_{\theta_{Pr}} c_0 \sim c_1 \ \{ \ w \ \}$. 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 θ_{Pr} and yields the expected specification $\theta_{Pr,St}(c_0, c_1) : \mathrm{StT}(\mathrm{BP})(A_0, A_1)$. The semantic judgment $\vDash_{\theta_{Pr,St}} c_0 \sim c_1 \ \{ \ w \ \}$ 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 modelling 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 $\mathrm{SD}(A)$ (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. [8] 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 $J : \mathcal{I} \to \mathcal{C}$, a monad relative to J (or J-relative monad) is a functor $M : \mathcal{I} \to \mathcal{C}$ equipped with "J-shifted" return and bind operations

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

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

$$\begin{aligned} \mathtt{bind}_{X,X}(\mathtt{return}_X) &= \mathrm{id}_{MX} \\ \mathtt{bind}_{X,Y}(k) \circ \mathtt{return}_X &= k \\ \mathtt{bind}_{X,Z}(\mathtt{bind}_{Y,Z}(l) \circ k) &= \mathtt{bind}_{Y,Z}(l) \circ \mathtt{bind}_{X,Y}(k) \end{aligned}$$

As a trivial example, any monad $M : C \to C$ can be seen as a relative monad over the identity functor Id_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} : chTy → Type over the inclusion functor.

Regarding specifications, relational specification monads W fail to form monads as well. Indeed, W : Type × Type → 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 dprod mapping two types to their product seen as a trivial preorder:

$$\begin{array}{rccc} \mathrm{dprod}: & \mathrm{Type} \times \mathrm{Type} & \to & \mathrm{PreOrder} \\ & A_0, A_1 & \mapsto & A_0 \times A_1. \end{array}$$

Specializing the definition of a relative monad with J = dprod, the `bind` and `return` operations of W take the following form:

$$\begin{array}{rl} \mathtt{return}^{\mathrm{W}}: & A_0 \times A_1 \to W(A_0, A_1) \\ \mathtt{bind}^{\mathrm{W}}: & (A_0 \times A_1 \to W(B_0, B_1)) \to W(A_0, A_1) \to W(B_0, B_1) \end{array}$$

To soundly model relational program logics, the $\mathtt{bind}^{\mathrm{W}}$ 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 [50, 51, 70]. 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 dprod : Type × Type → PreOrder.

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

$$\begin{array}{rl} \mathrm{dom}(\theta_{Pr}) \text{ is} & \mathrm{F}_{Pr} \times \mathrm{F}_{Pr} : \mathrm{chTy} \times \mathrm{chTy} \to \mathrm{Type} \times \mathrm{Type} \\ \mathrm{cod}(\theta_{Pr}) \text{ is} & \mathrm{BP} : \mathrm{Type} \times \mathrm{Type} \to \mathrm{PreOrder} \end{array}$$

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:

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

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

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

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

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

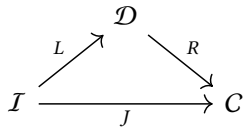
$$\theta\,(\mathtt{bind}^{M_0}\, f_0\, m_0, \mathtt{bind}^{M_1}\, f_1\, m_1) \leq \mathtt{bind}^{W}\,(\theta \circ (f_0, f_1))\;\theta\,(m_0, m_1) \tag{5}$$

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

In our development, relational effect observations $\theta : M_0 \times M_1 \longrightarrow W$ are defined as special cases of *lax morphisms between order-enriched relative monads*. We refer the interested reader to our formalization[7] 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 θ_{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 [56], 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 [8].

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

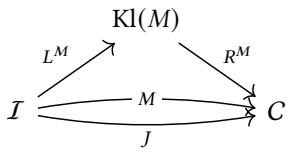


We say that L and R are J-relative left and right adjoints respectively ($L \ {}_J\dashv R$) if there exists a natural isomorphism $\forall (X : \mathcal{I})\ (Y : \mathcal{C}).\ \mathcal{D}(LX, Y) \cong \mathcal{C}(JX, RY)$. In that case, the composition RL turns out to be a J-relative monad and is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

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

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

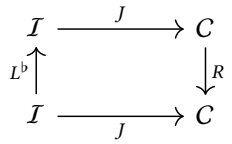
It is indeed a category exactly thanks to the monad laws of M. Moreover there exist functors L^M, R^M in the following configuration



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 $J, L^\flat, R$ in the following configuration:



An adjunction $\alpha : JL^\flat \ {}_J\dashv R$ 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 $\sigma : J \circ (- \times S) \ {}_J\dashv S \longrightarrow -$, which we suggestively

[7] https://github.com/SSProve/ssprove/blob/journal-submission/theories/Relational/OrderEnrichedCategory.v#L379

call "state-transforming adjunction". Note that the J-relative monad induced by this adjunction $X \mapsto S \to J(X \times S)$ is a J-shifted version of a standard state monad.

$$\begin{array}{ccc} \mathcal{I} & \xrightarrow{\quad J \quad} & \mathcal{C} \\ {\scriptstyle -\times S}\uparrow & & \downarrow{\scriptstyle S \to -} \\ \mathcal{I} & \xrightarrow[\quad J \quad]{} & \mathcal{C} \end{array}$$

Theorem 5.5 (Relative transformer). *Given a J-relative monad $M : \mathcal{I} \to \mathcal{C}$ "sitting" on a transforming adjunction $\alpha : JL^\flat \ {}_J \dashv R$, the composition $RML^\flat$ is also a J-relative monad. We call it the relative monad transformed by α and denote it as $\mathrm{T}_\alpha M$.*

Proof Sketch. We can factorize M through its Kleisli category as shown in Definition 5.3 to obtain $\mathrm{T}_\alpha M := RML^\flat = RR^M L^M L^\flat$ and observe that $L^M L^\flat \ {}_J \dashv RR^M$, meaning that $\mathrm{T}_\alpha M$ is the relative monad induced by the latter adjunction. □

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

Transforming lax morphisms. In order to transform lax morphisms of relative monads (such as θ_{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 $\theta : M \to W$ be a lax morphism of relative monads. Let α be a transforming adjunction for both M and W. Then θ induces a lax morphism between the Kleisli adjunctions of its domain and codomain:

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

$\mathrm{Kl}(\theta)$ can then be pasted with an appropriate cell to obtain a lax morphism between the transformed adjunctions, which ultimately induces a morphism $\mathrm{T}_\alpha \theta : \mathrm{T}_\alpha M \to \mathrm{T}_\alpha W$ between the transformed relative monads. Adding state to a relational effect observation θ now consists in applying the above with the state-transforming adjunction σ. This is how we can obtain $\mathrm{StT}(\theta_{Pr}) := \mathrm{T}_\sigma \theta_{Pr}$ in a modular way.

5.5 Comparing approaches to semantic models for relational program logics

We use the semantic framework of Maillard et al. [58] 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 modelling relational program logics.

The Foundational Cryptography Framework (FCF) [66] develops machine-checked proofs of cryptographic code in the Coq proof assistant. Computations are modelled 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, FCF distinguishes the type of simple computations `Comp` R with result value in R from the type of computations with access to some oracle `OracleComp` $I\, O\, R$. The latter provides operations to query an oracle with a given value in I and obtain results in O of an oracle call that are not found in simple computations. We rely on the genericity of free monads at the level of packages to encode both the simple computations and computations with oracles in a single uniform type of code, using the parameterized operations to provide oracle queries for instance. 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 in Coq in a project called XHL with respect to a model [81], parts of which have been merged into `mathcomp-analysis` [4]. 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 of Maillard et al. [58], 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 $\vDash_{\theta_{Pr,St}} c_0 \sim c_1 \{ w \}$ can seem abstract at first, it ultimately reduces to a direct formulation equivalent to the one underlying XHL.

CertiCrypt is a predecessor of EasyCrypt embedded directly in Coq and our foundational approach is closer in spirit to CertiCrypt than to its successor. CertiCrypt's Coq code employs an elaborate definition of probabilistic Relational Hoare Logic judgments in terms of approximate couplings, based on the ALEA library [11, 65], which directly offers support for bounding the distance between the distributions of two probabilistic programs. It also features a memory model distinguishing between local and global variables, providing the ability to give a direct semantics for first-order procedure calls with local state in the form of "stack frames". This memory model could be employed easily in our account of the state relative monad transformer.

CertiCrypt, EasyCrypt and FCF provide some support for unbounded while loops, ultimately relying at the semantic level on the fact that distributions take values in a complete lattice, namely the real numbers, and using standard fixpoint theorems to obtain an interpretation of arbitrary loops. Our semantic account could also support such an extension, as witnessed by the semantics Maillard et al. [58] construct for a simple Imp language with unbounded loops.

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 make efficient use of the factorisation provided by abstraction and bring this approach to a competitive level compared to the simpler direct approach of FCF [66] and XHL [81].

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 [40], and used for instance in the CryptoBox protocol [28]. 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 mechanized proof of KEM-DEM faithfully follows the single-instance security proof done on paper by Brzuska et al. [34, Section 4]. In fact, while conducting the proof in SSProve, we were able to find—in conjunction with the authors of [34]—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 parameterized over a type of symmetric keys and distribution keyD for generating them. In this respect we generalize [34], which uniformly samples over bit-strings of a given length. We give the KEY package in Figure 14.

```
package: KEY
mem:  k_loc : option key
```

```
Gen():
  k ← get k_loc
  assert k = ⊥ ;
  k <$ keyD
  put k_loc := Some k

Set(k):
  k' ← get k_loc
  assert k' = ⊥
  put k_loc := Some k

Get():
  k ← get k_loc
  assert (k ≠ ⊥) as kSome
  return (getSome k kSome)
```

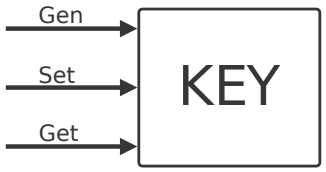


Fig. 14. KEY package

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^b is given by a core keying game $\mathtt{CK}^b$ with $\mathtt{import}(\mathtt{CK}^0) = \{\mathtt{Set}\}$, and $\mathtt{import}(\mathtt{CK}^1) = \{\mathtt{Gen}\}$, and $\mathtt{Get} \notin \mathtt{export}(\mathtt{CK}^b)$,

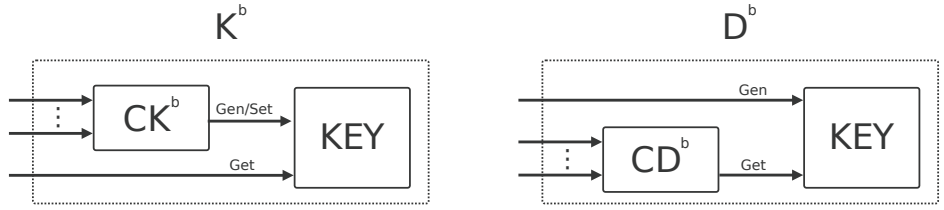


Fig. 15. Keying and keyed games

while a *keyed game* D^b is given by a core keyed game $\mathtt{CD}^b$ such that $\mathtt{import}(\mathtt{CD}^b) = \{\mathtt{Get}\}$, and $\mathtt{Gen} \notin \mathtt{export}(\mathtt{CD}^b)$. They are graphically represented in Figure 15 (where we write $\mathtt{Gen/Set}$ to indicate that the import is either $\mathtt{Gen}$ or $\mathtt{Set}$ depending on the secret bit b) and defined as follows:

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

The import and export interface conditions stated above ensure that these packages are meaningful.

As we will see, we will define the KEM package as a 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 pair K^{01} and a keyed game pair D^{01} as above, we have the following inequalities for any distinguisher $\mathcal{D}$:*

$$\begin{aligned} &\alpha((\mathtt{CK}^0 \parallel \mathtt{CD}^0) \circ \mathtt{KEY}, (\mathtt{CK}^1 \parallel \mathtt{CD}^1) \circ \mathtt{KEY})(\mathcal{D}) \leq \\ &\alpha(\mathtt{K}^{01})(\mathcal{D} \circ (\mathtt{ID}_{\mathtt{export}(\mathtt{CK})} \parallel \mathtt{CD}^0)) + \\ &\alpha(\mathtt{D}^{01})(\mathcal{D} \circ (\mathtt{CK}^1 \parallel \mathtt{ID}_{\mathtt{export}(\mathtt{CD})})) \end{aligned}$$

$$\begin{aligned} &\alpha((\mathtt{CK}^0 \parallel \mathtt{CD}^0) \circ \mathtt{KEY}, (\mathtt{CK}^0 \parallel \mathtt{CD}^1) \circ \mathtt{KEY})(\mathcal{D}) \leq \\ &\alpha(\mathtt{K}^{01})(\mathcal{D} \circ (\mathtt{ID}_{\mathtt{export}(\mathtt{CK})} \parallel \mathtt{CD}^0)) + \\ &\alpha(\mathtt{D}^{01})(\mathcal{D} \circ (\mathtt{CK}^1 \parallel \mathtt{ID}_{\mathtt{export}(\mathtt{CD})})) + \\ &\alpha(\mathtt{K}^{01})(\mathcal{D} \circ (\mathtt{ID}_{\mathtt{export}(\mathtt{CK})} \parallel \mathtt{CD}^1)) \end{aligned}$$

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 $\mathtt{CK}^0$ to $\mathtt{CK}^1$ in $(\mathtt{CK}^0 \parallel \mathtt{CD}^0) \circ \mathtt{KEY}$ by "pushing" $\mathtt{CD}^0$ to the left (i.e., using the reduction on top), meaning we obtain equal package $(\mathtt{ID}_{\mathtt{export}(\mathtt{CK})} \parallel \mathtt{CD}^0) \circ \mathtt{K}^0$. We then hop to $(\mathtt{ID}_{\mathtt{export}(\mathtt{CK})} \parallel \mathtt{CD}^0) \circ \mathtt{K}^1$, incurring the term $\alpha(\mathtt{K}^{01})(\mathcal{D} \circ (\mathtt{ID}_{\mathtt{export}(\mathtt{CK})} \parallel \mathtt{CD}^0))$ in the inequality, and then proceed back to $(\mathtt{CK}^1 \parallel \mathtt{CD}^0) \circ \mathtt{KEY}$ 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 ($\mathtt{CK}^1$ goes back to $\mathtt{CK}^0$), hence the extra term in the inequality. □

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.

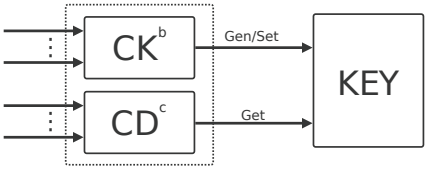


Fig. 16. Keying and keyed games combined

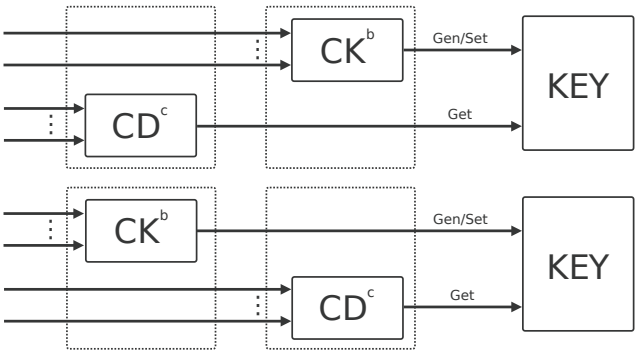


Fig. 17. Reductions for the keying and keyed games

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 tell which secret key correspond to which public key. We furthermore assume a symmetric key space `key` and an encrypted symmetric key space `ekey` together with distributions `keyD` and `ekeyD` on them.[8] 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 `0`), as well as a distribution on ciphers `cipherD`. In the original SSP paper [34], these distributions are uniform distributions and these types are described using bit-strings 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, η, is given by the following:

(1) η.kgen, a—state-preserving[9] and typically sampling—procedure which generates a valid public/secret key pair according to the `pkey_pair` relation;
(2) η.encap, a state-preserving procedure which takes a public key `pk` and generates a symmetric key together with its asymmetric encryption with `pk`;
(3) η.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 η.decap to the encrypted key returned by η.encap. This specification, and the specification of η.kgen are handled in our formalization using the diagonal of our probabilistic relational program logic, i.e., we ensure that property φ holds of c by verifying that the judgment $\forall \psi.\ \vDash \{\!\{\psi\}\!\}\ c \sim c\ \{\!\{(m_0', a_0), (m_1', a_1).\ \psi(m_0', m_1') \land a_0 = a_1 \land \varphi(a_0)\}\!\}$ holds.

A DEM, θ, is given by the following:

[8]The same `keyD` is used in the `KEY` package.
[9]Observationally, the state must be the same after execution of the procedure.

(1) θ.enc, a deterministic encryption function taking a symmetric key to turn a plaintext into a cipher;
(2) θ.dec, a deterministic decryption function.

Note that we do not need to know that θ.dec and θ.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.

For the remainder of this section, we assume that we are given a KEM η and a DEM θ. Using them we define the KEM^b and DEM^b games in Figure 18[10]. These games are then used respectively as core keying and keyed games in the $\mathsf{KEM\text{-}CCA}^{01}$ and $\mathsf{DEM\text{-}CCA}^{01}$ game pairs (represented in Figure 19):

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

We finally combine η and θ to form a public-key encryption (PKE) in the form the $\mathsf{PKE\text{-}CCA}^{01}$ game pair defined in Figure 20 and Figure 22.

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 $\mathsf{PKE\text{-}CCA}^{01}$, but which will eventually be composed sequentially with KEM, DEM and KEY to form an auxiliary game, featured in Figure 24 and defined as:

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

The security theorem that we formalize is then the following.

Theorem 6.2. *For every adversary to* $\mathsf{PKE\text{-}CCA}^{01}$ *and* AUX^{01} $\mathcal{A}$ *we have the following inequality:*

$$\begin{aligned}&\alpha(\mathsf{PKE\text{-}CCA}^{01})(\mathcal{A}) \leq \\ &\alpha(\mathsf{KEM\text{-}CCA}^{01})(\mathcal{A} \circ \mathsf{MOD\text{-}CCA} \circ (\mathsf{ID}_{\mathsf{export}(\mathsf{KEM})} \parallel \mathsf{DEM}^0)) + \\ &\alpha(\mathsf{DEM\text{-}CCA}^{01})(\mathcal{A} \circ \mathsf{MOD\text{-}CCA} \circ (\mathsf{KEM}^1 \parallel \mathsf{ID}_{\mathsf{export}(\mathsf{DEM})})) + \\ &\alpha(\mathsf{KEM\text{-}CCA}^{01})(\mathcal{A} \circ \mathsf{MOD\text{-}CCA} \circ (\mathsf{ID}_{\mathsf{export}(\mathsf{KEM})} \parallel \mathsf{DEM}^1))\end{aligned}$$

Note here that unfortunately, one detail of the proof leaks into the theorem statement: the adversary is also forbidden from using the state of the intermediary game pair AUX^{01}. Concretely, that means an adversary is not allowed to use k_loc in addition to the locations of $\mathsf{PKE\text{-}CCA}^{01}$.

Proof of Theorem 6.2. We use Lemma 2.2 to establish the inequality

$$\begin{aligned}&\alpha(\mathsf{PKE\text{-}CCA}^{01})(\mathcal{A}) \leq \\ &\alpha(\mathsf{PKE\text{-}CCA}^{0}, \mathsf{MOD\text{-}CCA} \circ (\mathsf{KEM}^0 \parallel \mathsf{DEM}^0) \circ \mathsf{KEY})(\mathcal{A}) + \\ &\alpha(\mathsf{MOD\text{-}CCA} \circ (\mathsf{KEM}^0 \parallel \mathsf{DEM}^0) \circ \mathsf{KEY}, \mathsf{MOD\text{-}CCA} \circ (\mathsf{KEM}^0 \parallel \mathsf{DEM}^1) \circ \mathsf{KEY})(\mathcal{A}) + \\ &\alpha(\mathsf{MOD\text{-}CCA} \circ (\mathsf{KEM}^0 \parallel \mathsf{DEM}^1) \circ \mathsf{KEY}, \mathsf{PKE\text{-}CCA}^{1})(\mathcal{A})\end{aligned} \tag{6}$$

which corresponds to the following if we fold the definition of AUX:

$$\begin{aligned}&\alpha(\mathsf{PKE\text{-}CCA}^{01})(\mathcal{A}) \leq \\ &\alpha(\mathsf{PKE\text{-}CCA}^{0}, \mathsf{AUX}^{0})(\mathcal{A}) + \\ &\alpha(\mathsf{MOD\text{-}CCA} \circ (\mathsf{KEM}^0 \parallel \mathsf{DEM}^0) \circ \mathsf{KEY}, \mathsf{MOD\text{-}CCA} \circ (\mathsf{KEM}^0 \parallel \mathsf{DEM}^1) \circ \mathsf{KEY})(\mathcal{A}) + \\ &\alpha(\mathsf{AUX}^{1}, \mathsf{PKE\text{-}CCA}^{1})(\mathcal{A})\end{aligned} \tag{7}$$

We then show that $\mathsf{PKE\text{-}CCA}^b$ and AUX^b are perfectly indistinguishable, using Theorem 2.4, i.e., we show they define equivalent procedures using an appropriate invariant. We detail the invariant

[10]We abuse **if** notation here to match with our examples in §2. Formally, c being set in the two branches can be understood as the whole **if** computing the value that is then stored in c.

```
package: KEM^b
mem:  pk_loc : option pkey
      sk_loc : option skey
      ek_loc : option ekey

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

Encap():
  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
  if b then Set(k) else Gen()
  return ek

Decap(ek'):
  sk ← get sk_loc
  assert sk ≠ ⊥ as skSome
  let sk := getSome sk skSome in
  ek ← get ek_loc
  assert ek ≠ Some ek'
  return η.decap(sk,ek')
```

```
package: DEM^b
mem:  c_loc : option cipher

Enc(msg):
  c ← get c_loc
  assert c = ⊥
  k ← Get()
  if b then
    c ← θ.enc(k,msg)
  else
    c ← θ.enc(k,0)
  put c_loc := Some c
  return c

Dec(c'):
  c ← get c_loc
  assert c ≠ Some c'
  k ← Get()
  return θ.dec(k,c')
```

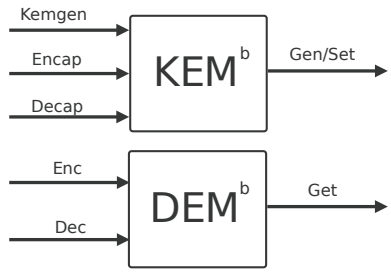


Fig. 18. KEM^b and DEM^b games

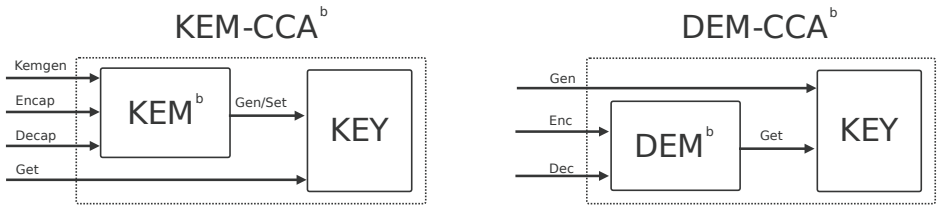


Fig. 19. KEM-CCA^b and DEM-CCA^b games

and the code comparisons in Appendix A. We now proceed with rest of the proof of Theorem 6.2.

```
package: PKE-CCA^b
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, 0)
  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
  c ← get c_loc
  assert (
    (ek, c) ≠ (Some ek', Some c')
  )
  k ← η.decap(sk, ek')
  return θ.dec(k, c')
```

Fig. 20. PKE-CCA^b 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
  c ← get c_loc
  assert (
    (ek, c) ≠ (Some ek', Some c')
  )
  if ek = Some ek' then
    msg ← Dec(c')
  else
    k' ← Decap(ek')
    msg ← θ.dec(k', c')
  return msg
```

Fig. 21. MOD-CCA game

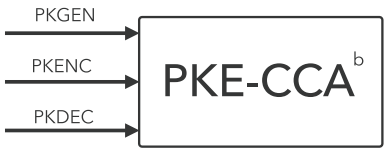


Fig. 22. PKE-CCA^b game graphically

Fig. 23. MOD-CCA package graphically

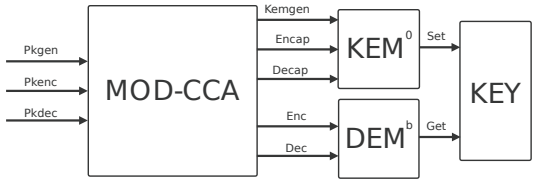


Fig. 24. AUX^b game construction

Since PKE-CCA^b is perfectly indistinguishable from AUX^b, the inequality (7) simplifies to

$$\alpha(\mathsf{PKE\text{-}CCA}^{01})(\mathcal{A}) \leq \alpha(\mathsf{MOD\text{-}CCA} \circ (\mathsf{KEM}^0 \parallel \mathsf{DEM}^0) \circ \mathsf{KEY}, \mathsf{MOD\text{-}CCA} \circ (\mathsf{KEM}^0 \parallel \mathsf{DEM}^1) \circ \mathsf{KEY})(\mathcal{A}). \quad (8)$$

Using Lemma 2.3 we replace the right-hand side to derive

$$\alpha(\mathsf{PKE\text{-}CCA}^{01})(\mathcal{A}) \leq \alpha((\mathsf{KEM}^0 \parallel \mathsf{DEM}^0) \circ \mathsf{KEY}, (\mathsf{KEM}^0 \parallel \mathsf{DEM}^1) \circ \mathsf{KEY})(\mathcal{A} \circ \mathsf{MOD\text{-}CCA}). \quad (9)$$

This (right-hand) upper bound corresponds to an instance of the left-hand side of the second inequality of Lemma 6.1 using $\mathcal{A} \circ \mathsf{MOD\text{-}CCA}$ as distinguisher, meaning we have the inequality

$$\begin{aligned} &\alpha((\mathsf{KEM}^0 \parallel \mathsf{DEM}^0) \circ \mathsf{KEY}, (\mathsf{KEM}^0 \parallel \mathsf{DEM}^1) \circ \mathsf{KEY})(\mathcal{A} \circ \mathsf{MOD\text{-}CCA}) \leq \\ &\alpha(\mathsf{KEM\text{-}CCA}^{01})(\mathcal{A} \circ \mathsf{MOD\text{-}CCA} \circ (\mathsf{ID}_{\mathsf{export}(\mathsf{KEM})} \parallel \mathsf{DEM}^0)) + \\ &\alpha(\mathsf{DEM\text{-}CCA}^{01})(\mathcal{A} \circ \mathsf{MOD\text{-}CCA} \circ (\mathsf{KEM}^1 \parallel \mathsf{ID}_{\mathsf{export}(\mathsf{DEM})})) + \\ &\alpha(\mathsf{KEM\text{-}CCA}^{01})(\mathcal{A} \circ \mathsf{MOD\text{-}CCA} \circ (\mathsf{ID}_{\mathsf{export}(\mathsf{KEM})} \parallel \mathsf{DEM}^1)). \end{aligned} \quad (10)$$

We thus conclude using transitivity of (9) and (10). □

7 CASE STUDY: Σ-PROTOCOLS

Σ-protocols form an important class of zero-knowledge protocols [42, 49]. A Σ-protocol is defined on a relation **R** for which a prover, in zero-knowledge, can prove it knows a witness w for a public statement h such that the relations **R** h w holds.

In this section, we show how we can define the class of Σ-protocols in SSProve. We then prove security of a transformation converting a Σ-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 [75] to be a member of the class of Σ-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.

The general flow of a Σ-protocol can be seen in Figure 25. First, the prover uses the secret information w to compute a message a which is sent to the verifier. Second, the verifier samples a challenge e uniformly at random from some challenge space and sends it to the prover. Third, the prover computes a response z based on the secret information, the message, and the challenge.

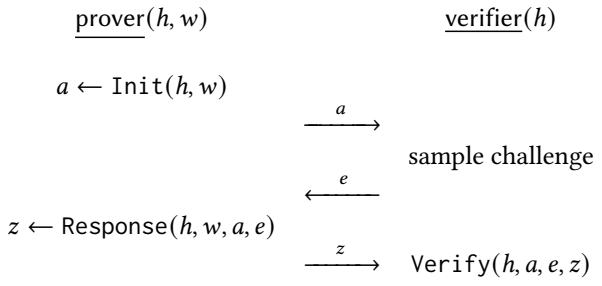


Fig. 25. Σ-Protocol overview

The response is then sent to the verifier. Finally, the verifier takes the public information, message, challenge, and response and checks whether 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 Σ-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 scheme

Σ-protocols have been formalized before. Butler et al. [36] formalize a number of Σ-protocols in CryptHOL, and prove compositional properties of these protocols. Sidorenco et al. [77] formalize Σ-protocols in EasyCrypt, and combine them with a formalization of multi-party computation (MPC) to formalize a, so-called, MPC-in-the-head protocol.

For our representation of Σ-protocols, we build on both these formalizations to define the SIGMA scheme and its corresponding security properties.

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

Additionally, we assume uniform distributions, *challengeD* and *witnessD*, for sampling challenges and witnesses of the relation, respectively.

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

For any game which depends on a Σ scheme, we denote the scheme as follows: P_S denotes the P game depending on the Σ scheme S. For brevity, the scheme is sometimes omitted from the notation.

Definition 7.1. The *special honest-verifier zero-knowledge* security of an instantiation of a SIGMA scheme S against adversary $\mathcal{A}$ is the advantage $\alpha(\mathsf{SHVZK}^0_S, \mathsf{SHVZK}^1_S)(\mathcal{A})$.

Definition 7.2. The *special soundness* of an instantiation of a SIGMA scheme S against adversary $\mathcal{A}$ is the advantage $\alpha(\mathsf{SOUND}^0_S, \mathsf{SOUND}^1_S)(\mathcal{A})$.

```
package: SHVZK⁰
mem:
Main(h, w, e):
  assert (R h w)
  (a, s) ← Σ.Init(h, w)
  z ← Σ.Response(s, h, w, a, e)
  return (h, a, e, z)
```

```
package: SHVZK¹
mem:
Main(h, w, e):
  assert (R h w)
  (a, z) ← Σ.Simulate(h, e)
  return (h, a, e, z)
```

Fig. 26. SHVZK^b game

```
package: SOUND⁰
mem:
Extract(h, a, e, e', z, z'):
  v ← Σ.Verify(h, a, e, z)
  v' ← Σ.Verify(h, a, e', z')
  if ((e ≠ e') && v && v') then
    w' ← Σ.Extract(h, a, e, e', z,
     z')
    return (R h w')
  else
    return false
```

```
package: SOUND¹
mem:
Extract(h, a, e, e', z, z'):
  v ← Σ.Verify(h, a, e, z)
  v' ← Σ.Verify(h, a, e', z')
  return (e ≠ e') && v && v'
```

Fig. 27. SOUND^b game

7.2 Commitment Schemes from Σ-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.

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 parameterized 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 30 and Figure 31.

Following the presentation of Damgaard [42], we show how our `SIGMA` scheme with related security games can be used to construct a commitment scheme. This result is mostly of theoretical interest, and has previously been formalized in CryptHOL [36]. We formalize the result in SSProve to experiment with leveraging the package laws to build one protocol from another, and compare this result with the construction in CryptHOL in Section 7.2.1.

The key component of this transformation is the `COM` package shown in Figure 29. Here, `COM` is parameterized by the `SIGMA` scheme. Moreover, `COM` is meant to be composed with the `KEY` package, which is responsible for distributing the public and secret keys between the parties, and as such imports `Init` and `Get`. The `KEY` package is shown in Figure 28. `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 Σ-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 Σ-protocol.
(3) `Ver`, which takes the commitment and the opening information and verifies their consistency. This, again, is dependent on the underlying Σ-protocol.

In this transformation, the types of the underlying Σ-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 Σ-protocol.

First, in Theorem 7.4 we show that the construction is hiding with its security bounded by the Special Honest-Verifier Zero-Knowledge property of the underlying Σ-protocol.

Theorem 7.4. *For any instantiation of the* `SIGMA` *scheme, the commitment scheme given by* `COM`$\circ$`KEY`*, where the adversary is given oracle access to the* `KEY` *package, is hiding with the following advantage:*

$$\begin{aligned}
&\alpha((\mathsf{HIDE}^0 \circ \mathsf{COM} \circ \mathsf{KEY}) \parallel \mathsf{KEY}, (\mathsf{HIDE}^1 \circ \mathsf{COM} \circ \mathsf{KEY}) \parallel \mathsf{KEY})(\mathcal{A}) \leq \\
&\alpha((\mathsf{HIDE}^0 \circ \mathsf{AUX} \circ \mathsf{SHVZK}^0) \parallel \mathsf{KEY}, (\mathsf{HIDE}^1 \circ \mathsf{AUX} \circ \mathsf{SHVZK}^1) \parallel \mathsf{KEY})(\mathcal{A}) + \\
&\alpha(\mathsf{SHVZK}^{01})(\mathcal{A} \circ (\mathsf{HIDE}^0 \circ \mathsf{AUX} \parallel \mathsf{KEY})) + \\
&\alpha(\mathsf{SHVZK}^{01})(\mathcal{A} \circ (\mathsf{HIDE}^1 \circ \mathsf{AUX} \parallel \mathsf{KEY}))
\end{aligned}$$

where `AUX` *is the package given by inlining* (`SIGMA` $\parallel$ `KEY`) *into* `COM` *and replacing the call to the* Σ*-protocol simulator with* $\mathsf{SHVZK}^1.\mathit{Main}$*. Note that* $\mathsf{SHVZK}^1.\mathit{Main}$ *requires both a public statement and a witness as its arguments. Since* `COM` *calls* $\mathsf{KEY}.\mathit{Init}$ *the witness exists within the package* `COM` $\circ$ `KEY`*. The* $\mathsf{SHVZK}^1.\mathit{Main}$ *procedure is called with the statement and witness produced by* `KEY`*.*

Proof of Theorem 7.4. First, let $\mathcal{D} = \mathcal{A} \circ (\mathsf{ID}_{exports(\mathsf{HIDE})} \parallel \mathsf{KEY})$. Applying Lemma 2.2 we obtain:

```
package: KEY
mem:  setup_loc : option bool
      h_loc : option statement
      w_loc : option witness
```

```
Init():
  b ← get setup_loc
  assert (b ≠ ⊥)
  w <$ witnessD
  let h := g^w in
  assert (R h w)
  put h_loc := Some h
  put w_loc := Some w
  put setup_loc := Some false
  return ()

Get():
  assert (b == Some false)
  h ← get h_loc
  return h
```

Fig. 28. KEY package

```
package: COM
mem:
```

```
Commit(m):
  Init()
  h ← Get()
  (c, o) ← Σ.Simulate(h, m)
  return (c, o)

Ver(c, m, o):
  h ← Get()
  v ← Σ.Verify(h, c, m, o)
  return v
```

Fig. 29. COM package

$$
\begin{aligned}
&\alpha((\mathsf{HIDE}^0 \circ \mathsf{COM} \circ \mathsf{KEY}) \parallel \mathsf{KEY}, (\mathsf{HIDE}^1 \circ \mathsf{COM} \circ \mathsf{KEY}) \parallel \mathsf{KEY})(\mathcal{D}) \leq \\
&\alpha((\mathsf{HIDE}^0 \circ \mathsf{COM} \circ \mathsf{KEY}) \parallel \mathsf{KEY}, \mathsf{HIDE}^0 \circ \mathsf{AUX} \circ \mathsf{SHVZK}^1)(\mathcal{D}) + \\
&\alpha(\mathsf{HIDE}^0 \circ \mathsf{AUX} \circ \mathsf{SHVZK}^1, \mathsf{HIDE}^0 \circ \mathsf{AUX} \circ \mathsf{SHVZK}^0)(\mathcal{D}) + \\
&\alpha(\mathsf{HIDE}^0 \circ \mathsf{AUX} \circ \mathsf{SHVZK}^0, \mathsf{HIDE}^1 \circ \mathsf{AUX} \circ \mathsf{SHVZK}^0)(\mathcal{D}) + \\
&\alpha(\mathsf{HIDE}^1 \circ \mathsf{AUX} \circ \mathsf{SHVZK}^0, \mathsf{HIDE}^1 \circ \mathsf{AUX} \circ \mathsf{SHVZK}^1)(\mathcal{D}) + \\
&\alpha(\mathsf{HIDE}^1 \circ \mathsf{AUX} \circ \mathsf{SHVZK}^1, (\mathsf{HIDE}^1 \circ \mathsf{COM} \circ \mathsf{KEY}) \parallel \mathsf{KEY})(\mathcal{D})
\end{aligned}
$$

```
package: HIDE^0
mem:
Hide(m1, m2):
  k <$ uniform bool
  if k then
    (c, o) ← Commit(m1)
  else
    (c, o) ← Commit(m2)
  return c
```

```
package: HIDE^1
mem:
Hide(m1, m2):
  m <$ challengeD
  (c, o) ← Commit(m)
  return c
```

Fig. 30. HIDE^b game

```
package: BIND
mem:
Bind(c, m, o, m', o'):
  v ← Ver(c, m, o)
  v' ← Ver(c, m', o')
  return (m ≠ m') && v && v' % TW:
     Here it was e ≠ e' which I
    changed to m/m' hoping it's
    correct.
```

Fig. 31. BIND Package

```
package: BIND-Aux
mem:
Bind(c, m, o, m', o'):
  h ← Get()
  b ← Extract(h, c, m, m', o, o')
  return b
```

Fig. 32. BIND-AUX Package

The advantage of moving to and from the AUX package is 0, since AUX consists of COM where the KEY package inlined and the simulator is replaced with calls to the SHVZK package.

By Lemma 2.3:

$$\alpha(\mathsf{HIDE}^0 \circ \mathsf{AUX} \circ \mathsf{SHVZK}^1, \mathsf{HIDE}^0 \circ \mathsf{AUX} \circ \mathsf{SHVZK}^0)(\mathcal{D}) = \alpha(\mathsf{SHVZK}^1, \mathsf{SHVZK}^0)(\mathcal{D} \circ \mathsf{HIDE}^0 \circ \mathsf{AUX}).$$

This gives us one of the bounds we are looking for. Similarly, the other SHVZK bound is obtained from $\alpha(\mathsf{HIDE}^1 \circ \mathsf{AUX} \circ \mathsf{SHVZK}^0, \mathsf{HIDE}^1 \circ \mathsf{AUX} \circ \mathsf{SHVZK}^1)(\mathcal{D})$.

This leaves us with $\alpha(\mathsf{HIDE}^0 \circ \mathsf{AUX} \circ \mathsf{SHVZK}^0, \mathsf{HIDE}^1 \circ \mathsf{AUX} \circ \mathsf{SHVZK}^0)(\mathcal{D})$, which is the last bound of Theorem 7.4. □

The binding property states that it must be infeasible for an adversary to produce two openings to the same commitments. Usually, this property is stated as: the probability of $\mathsf{BIND} \circ \mathsf{COM}$ returning true, for any input, is sufficiently small. In SSProve, we do not have a unary logic to express such a statement. Rather, we prove in our relational logic that any adversary which makes $\mathsf{BIND} \circ \mathsf{COM}$ return true, can be used to construct a program that outputs a witness of the relation. The package using the adversary to extract a witness for the relation is given by Figure 32. If a Σ-protocol is used to construct the commitment scheme, then the binding game is infeasible to win, because it is infeasible to compute the witness from the statement in a Σ-protocol.

Theorem 7.5. *For any instantiation of the Σ scheme, the probability of winning the binding game or computing the witness from the statement only differ by the probability of* Σ.Extract *outputting a valid witness for the relation:*

$$\alpha(\mathsf{BIND} \circ (\mathsf{COM} \circ \mathsf{KEY}), \mathsf{BIND\text{-}Aux} \circ \mathsf{SOUND}^0)(\mathcal{A} \circ (\mathsf{ID}_{exports(\mathsf{BIND})} \parallel \mathsf{KEY})) \leq \alpha(\mathsf{SOUND})(\mathcal{A} \circ ((\mathsf{BIND\text{-}AUX} \circ \mathsf{KEY})) \parallel \mathsf{KEY}).$$

To prove Theorem 7.5, we use Lemma 2.2 and Lemma 2.3 to replace the commitments with Σ-protocol transcripts, via SOUND^1.

7.2.1 Formalization in CryptHOL. The construction deriving commitment schemes from Σ-protocols has also been formalized in CryptHOL [36]. In their definition, the commitment scheme construction is dependent on the types needed to instantiate a Σ-protocol. The proofs of hiding and binding are then quantified over any secure Σ-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 Σ.*Response* and Σ.*Verify* procedures of the Σ-protocol terminate on all inputs.

In contrast, our security bounds defined in Theorem 7.4 and Theorem 7.5 make no assumptions on the underlying Σ-protocol other than that it implements 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 Σ-protocol.

The imperfect game hops are justified by the package theorems 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 [36]. Namely, if we adopt the same assumption, then the proof of the security bounds can be done entirely within 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 [75] is parameterized over a cyclic group $(\mathcal{G}, \star)$ with q elements generated by g. Schnorr's protocol is a Σ-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 $z \in \mathbb{Z}_q$. Challenges are sampled from a uniform distribution over $\mathbb{Z}_q$.

The protocol is implemented as an SSP package as shown in Figure 33. In particular, the package exports match the expected imports of our Σ-protocol security statements.

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

$$\alpha(\mathsf{SHVZK}^{01}_{\mathsf{SCHNORR}})(\mathcal{A}) = 0$$

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 1. Since the SHVZK^b package does not use state and the procedures of Σ scheme are stateless we use equality of heaps as our invariant.

When comparing the code in Table 1 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.

```
Σ.Init(h, w):
  r <$ witnessD
  return (g^r, r)

Σ.Response(s, h, w, a, e):
  return s + e * w

Σ.Verify(h, w, a, e, z):
  return g^z = a * h^e

Σ.Simulator(h, e):
  z <$ witnessD
  return (h, g^z * h^-e, e, z)

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

Fig. 33. SCHNORR Σ-scheme

Table 1. Code comparison of SHVZK^0 and SHVZK^1 with Schnorr's protocol

SHVZK^0.Main(h, w, e)	SHVZK^1.Main(h, w, e)
assert h = g^w	assert h = g^w
r <$ witnessD	z <$ witnessD
a ← g^r	a ← g^z * h^-e
z ← r + e * w	
return (h, a, e, z)	return (h, a, e, z)

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 $f : x \mapsto x + e \cdot w$ and moving all constants into the return statement we obtain: $(h, g^r, e, r + e \cdot w) \stackrel{?}{=} (h, g^{f(z)} \cdot h^{-e}, e, f(r))$. We solve the goal with the following equality:

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

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

Lemma 7.7 (Schnorr Special-Soundness). *For all adversaries $\mathcal{A}$ we have the following equality:*

$$\alpha(\mathsf{SOUND}^{01}_{\mathsf{SCHNORR}})(\mathcal{A}) = 0$$

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 2. Since

Table 2. Code comparison of SOUND^0 and SOUND^1 with Schnorr's protocol

$(\mathsf{SOUND}^0.\mathsf{Extract}(h, a, e, e', z, z')$	$(\mathsf{SOUND}^1.\mathsf{Extract}(h, a, e, e', z, z')$
$v \leftarrow g^z = a * h^e$	$v \leftarrow g^z = a * h^e$
$v' \leftarrow g^{z'} = a * h^{e'}$	$v' \leftarrow g^{z'} = a * h^{e'}$
if $((e \neq e')$ && v && $v')$ then	
$w' \leftarrow (z - z') / (e - e')$	
return $(h = g^{w'})$	
else	
return false	return $((e \neq e')$ && v && $v')$

Table 3. Code comparison of $\mathsf{HIDE}^0 \circ \mathsf{COM}_{\mathsf{SCHNORR}} \circ \mathsf{KEY}$ and $\mathsf{HIDE}^1 \circ \mathsf{COM}_{\mathsf{SCHNORR}} \circ \mathsf{KEY}$

$(\mathsf{HIDE}^0 \circ \mathsf{COM}_{\mathsf{SCHNORR}} \circ \mathsf{KEY}).\mathsf{Hide}(m1, m2)$	$(\mathsf{HIDE}^1 \circ \mathsf{COM}_{\mathsf{SCHNORR}} \circ \mathsf{KEY}).\mathsf{Hide}(m1, m2)$
k <$ uniform bool	
Init()	Init()
h ← Get()	h ← Get()
r <$ witnessD	m <$ challengeD
if k then	
a ← g^r	a ← g^r
z ← r + m1 * w	z ← r + m * w
put e_loc := Some m1	put e_loc := Some m
else	
a ← g^r	
z ← r + m2 * w	
put e_loc := Some m1	
put z_loc := Some z	put z_loc := Some z
return a	return a

neither Schnorr's protocol nor the SOUND^b package use state, we can use equality of heaps as our invariant.Both sides are given the same inputs so the two programs are indistinguishable after the two verifications. The if-statement in SOUND^0 is equal to the return value of SOUND^1. Hence, the two programs are indistinguishable if $h = g^{w'}$. We show this indeed holds:

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

□

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 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 all adversaries $\mathcal{A}$ we get the following equality for the commitment scheme instantiated from Schnorr's protocol:*

$$\alpha(\mathsf{HIDE}^{01} \circ \mathsf{COM}_{\mathsf{SCHNORR}} \circ \mathsf{KEY})(\mathcal{A}) = 0$$

Proof of Theorem 7.8. We use Theorem 7.4 to obtain:

$$\alpha((\mathsf{HIDE}^0 \circ \mathsf{COM} \circ \mathsf{KEY}) \parallel \mathsf{KEY}, (\mathsf{HIDE}^1 \circ \mathsf{COM} \circ \mathsf{KEY}) \parallel \mathsf{KEY})(\mathcal{A}) \leq$$
$$\alpha((\mathsf{HIDE}^0 \circ \mathsf{AUX} \circ \mathsf{SHVZK}^0) \parallel \mathsf{KEY}, (\mathsf{HIDE}^1 \circ \mathsf{AUX} \circ \mathsf{SHVZK}^1) \parallel \mathsf{KEY})(\mathcal{A}) +$$
$$\alpha(\mathsf{SHVZK})(\mathcal{A} \circ (\mathsf{HIDE}^0 \circ \mathsf{AUX} \parallel \mathsf{KEY})) +$$
$$\alpha(\mathsf{SHVZK})(\mathcal{A} \circ (\mathsf{HIDE}^1 \circ \mathsf{AUX} \parallel \mathsf{KEY}))$$

From Lemma 7.6 we get the two terms involving the advantage on the SHVZK game is 0.

$$\alpha((\mathsf{HIDE}^0 \circ \mathsf{COM} \circ \mathsf{KEY}) \parallel \mathsf{KEY}, (\mathsf{HIDE}^1 \circ \mathsf{COM} \circ \mathsf{KEY}) \parallel \mathsf{KEY})(\mathcal{A}) \leq$$
$$\alpha((\mathsf{HIDE}^0 \circ \mathsf{AUX} \circ \mathsf{SHVZK}^0) \parallel \mathsf{KEY}, (\mathsf{HIDE}^1 \circ \mathsf{AUX} \circ \mathsf{SHVZK}^1) \parallel \mathsf{KEY})(\mathcal{A})$$

To finish the proof we show that $\mathsf{HIDE}^0 \circ \mathsf{AUX} \circ \mathsf{SHVZK}^0$ and $\mathsf{HIDE}^1 \circ \mathsf{AUX} \circ \mathsf{SHVZK}^0$ 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 3.

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 3, 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 one-sided rules for writing memory. With this we have the same program on both sides up 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. □

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 [13]. In this section we survey the closest related work in this space.

CertiCrypt [22] 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 [66] is a more recent foundational Coq framework for cryptographic proofs that was used to verify the HMAC implementations in OpenSSL [27] and mbedTLS [85]. 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 could also allow a more modular treatment of programs exhibiting effects of different nature such as communications with an external process [58]. Building a formalization of SSP on top of FCF may also be possible in principle though, and maybe even more interesting in practice could be rebasing SSProve on a simpler but less modular semantic model in the style of FCF or XHL [81] (as discussed in §5.5).

EasyCrypt [17, 20] is a proof assistant and verification tool specifically designed for game-based cryptographic proofs and built from scratch. This state-of-the-art tool has been used, for instance, to prove security for Amazon Web Services' Key Management Service [6]. 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 [14], as well as other abstraction mechanisms such as "theories". The default way of mechanizing proofs in EasyCrypt is, however, quite different than that of SSP. The EasyCrypt abstraction mechanisms were designed for allowing reuse of code and theorems [17], but they are rarely used at the moment to provide a modular high-level structure to cryptographic reduction proofs in the style of SSP [43].

In concurrent work, Dupressoir et al. [43] show that they can code up in EasyCrypt an SSP-style multi-instance security proof for the Cryptobox [28] KEM-DEM [40] construction, and they discuss the various trade-offs they had to navigate and the strengths and shortcomings of EasyCrypt for formalizing such SSP-style proofs. Their multi-instance setting comes with specific challenges that do not appear in our mechanized proof from §6, such as the adversarial ability to create corrupt key instances. Yet despite the different setting, the high-level structure of the proofs is similar. A consequence of the faithfulness of our mechanization of the single-instance KEM-DEM proof of Brzuska et al. [34, Section 4] is that we could discover a flaw in the original theorem statement and paper proof, which emphasizes the benefits of mechanization.

Beyond this case study, we focus on providing a *general framework* for SSP proofs, including formal definitions of SSP packages, their composition, and the corresponding algebraic laws. Dupressoir et al. [43] show instead *informally*, in the context of their example, how SSP concepts can be mapped to existing features in EasyCrypt. In particular, games are mapped to modules, and packages with imports are mapped to ML-style functors [62], i.e., modules parameterized by other modules. Modules and functors are not fully first class in EasyCrypt: while it is possible to quantify over modules, formulas cannot talk about module equality [80]. As a result, SSP-style laws cannot be stated or proven, and one has to manually code up all the functors involved in the proof, and argue about equality of the individual procedures instead.

SSP also has a notion of parameterized packages, in particular packages parameterized by crypto schemes or multi-instance packages [34, Section 5]. On the one hand, in SSProve we can easily parametrize packages over parameters like crypto schemes using the expressive abstraction mechanisms of Coq's functional programming language, such as functions returning packages. By contrast, Dupressoir et al. do this using the module-level abstraction mechanisms of EasyCrypt, choosing between functors and theories on a case by case basis [43, pg. 7]. On the other hand, multi-instance packages in SSP are regular packages for which the names of the procedures (defined or used by the package) are themselves indexed by natural numbers. To support multi-instance packages in SSProve a small technical change is needed, we can parametrize both the packages' memory locations and procedure names by an offset in order to make each instance distinct, which some of the authors have already tried out in ongoing work on formally connecting SSProve and Jasmin [5]. By contrast, Dupressoir et al. [43] represent a multi-instance package as a single regular module (or functor) in EasyCrypt, having as state a map from instance indices to the actual state of each instance, and where the procedures also take an explicit instance index as argument.

SSProve also includes an `assert` operation, with a simple and clearly defined semantics: assertion failure samples from the empty probability subdistribution (the same as entering an infinite loop in EasyCrypt). While this is not the only choice [25], this formalizes our understanding of the following informal convention from the paper introducing SSP [34]: "all our definitions and theorems apply only to code that never violates assertions." By contrast, Dupressoir et al. [43], manually encode an "oracle silencing" semantics for assertion failures [72] in their case study, but without formally

providing an assert construct, and while calling out improving the conciseness of their formal definitions as future work.

SSProve also faithfully follows the SSP model for memory initialization, allowing to express SSP proofs more naturally. Using default initial values for implicitly initializing all state variables allows us to define the notion of distinguishing advantage in terms of running on an initial memory, as done in SSP. By contrast, Dupressoir et al. [43] use a mix of explicit initialization and logical preconditions to restrict the memories considered to those which are properly initialized. Finally, one aspect in which we took a similar design decision in SSProve to Dupressoir et al. [43], is the absence of the implicit α-renaming conventions and pervasive state separation of Brzuska et al. [34]. We instead reason about concrete locations and explicitly require state separation only between adversaries and the games with which they are composed.

EasyUC [39] aims to address the lack of composability in game-based proofs by formalizing the Universal Composability (UC) framework [37] 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 [38] 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. [14] add automatic complexity analysis to EasyCrypt and use it for another formalization of UC. ILC [54] is a process calculus modelling 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. SSP was in part inspired by the UC framework, but focuses on making game-based proofs more modular and scalable, without targeting universal composability. A more precise comparison between SSP and UC proofs would be interesting, but out of scope for the current paper. Recent work by Brzuska and Oechsner [35] and Brzuska et al. [33] indicates that SSP can also be relevant for simulation-based security.

CryptHOL [23] is a foundational framework for game-based proofs that established a connection between relational parametricity and coupling, the main workhorse of pRHL, to achieve automation in the Isabelle/HOL proof assistant. CryptHOL 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 the elegance of CryptHOL's formalization of ElGamal or PRF-based encryption. CryptHOL [55] has been also used to formalize Constructive Cryptography [59] (an instance of Abstract Cryptography [60]), another composable framework that inspired SSP, and the example of a one-time pad. CryptHOL converters are similar to SSP packages, however, there are certain distinctions that we discuss here: To begin, converters combine all their procedures into a single resumption-like value, resulting in simpler interfaces consisting of a dependent pair (A, B), where A denotes a global input type and B a global output type of the bundled procedures, and an additional invariant is maintained to ensure that outputs correspond to specific inputs. Procedures in SSP packages are written using a free monad that captures the probabilistic operations without evaluating them, and only at a later stage these operations are interpreted, whereas converters are built directly over the probabilistic subdistributions monad. Similarly, SSP packages utilize uninterpreted operations for stateful operations, whereas converters keep a hidden state using the coinductive structure of resumptions. CryptHOL provides a bisimilarity notion that can be used to prove perfect indistinguishability, similar to our Theorem 2.4, but in addition, they also provide a bisimulation-style proof rule for establishing trace equivalence, which as they show is needed in certain cryptographic arguments.

Regarding automation, both Isabelle/CryptHOL and EasyCrypt are based on classical HOL and provide powerful SMT-based automation. A detailed comparison between the HOL-systems and

systems like Coq based on dependent type theory is out of scope for the present paper. We merely observe that huge formalizations have been carried out in both traditions. On the narrow topic of SMT-based automation, we mention this is also being developed for Coq [10, 41], and any progress on those projects could also profit SSProve. Another key part of the automation in EasyCrypt is the auto tactic, which tries a to apply rules of the program logic based on the structure of the code. We have a similar tactic in SSProve, which moreover, is easily extensible due to Coq's tactic language.

IPDL [64] 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 of IPDL to other composable frameworks has not been worked out, and is out of scope for the current paper. IPDL uses the same one-time pad example as Constructive Cryptography [59], and in fact their compositional formalism (though not explicitly stated that way) seems closer to Constructive Cryptography than UC.

Packages have been motivated by ML modules [73]. 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 [79] 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 [29, 30, 33, 45] were the main inspiration for the SSP methodology and it would be an interesting challenge to scale SSProve to mechanizing such large 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 Jasmin [5, 7], VST [9], or FiatCrypto [44]. Some of the authors are in fact working to integrate SSProve with Jasmin [5, 7] by defining a translation from the representation of Jasmin programs in Coq to SSProve programs, while provably preserving semantics. This allows us to formally connect in Coq security proofs in SSProve to the assembly code produced by the Jasmin verified compiler.

An alternative would be to port SSProve to F* [82], where at least functional correctness can be verified at that scale. Still many challenges would remain, including extending F* to probabilistic verification, giving F* modules first-class status, 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. [52] have implemented vanilla SSP packages in F* and attempted to automate state-separating proofs based on a library for partial setoids.

Finally, our formalization of Σ-protocols has been used recently to prove the security of the OpenVoteNetwork smart contract [78].

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 also grateful to Markulf Kohlweiss for pointing out a bug in our modelling of public-key encryption schemes. We were able to quickly fix the security definition and proof of ElGamal, as discussed in §2.4. We also thank Brzuska et al. [34] for their prompt fix of the informal proof of security for KEM-DEM. Finally we are grateful to the anonymous reviewers of CSF and TOPLAS for their detailed and helpful 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*, by the German Federal Ministry of Education and Research BMBF (grant 16K15K042, project 6GEM) 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 Hriţcu, 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] Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer, and Kathrin Stark. 2019. POPLMark reloaded: Mechanizing proofs by logical relations. *J. Funct. Program.* 29 (2019), e19. https://doi.org/10.1017/S0956796819000170

[3] 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.

[4] 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

[5] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. 2017. Jasmin: High-Assurance and High-Speed Cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 1807–1823. https://doi.org/10.1145/3133956.3134078

[6] 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

[7] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. 2020. The Last Mile: High-Assurance and High-Speed Cryptographic Implementations. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 965–982. https://doi.org/10.1109/SP40000.2020.00028

[8] 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

[9] 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

[10] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. 2011. A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*. 135–150.

[11] Philippe Audebaud and Christine Paulin-Mohring. 2006. Proofs of Randomized Algorithms in Coq. In *Mathematics of Program Construction*. Springer, 49–68.

[12] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. 2005. Mechanized Metatheory for the Masses: The PoplMark Challenge. In *18th International Conference on Theorem Proving in Higher Order Logics*. 50–65. https://doi.org/10.1007/11541868_4

[13] 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*.

[14] Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, and Pierre-Yves Strub. 2021. Mechanized Proofs of Adversarial Complexity and Application to Universal Composability. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 2541–2563. https://doi.org/10.1145/3460120.3484548

[15] R. Barnes, B. Beurdouche, R. Robert, J. Millican, E. Omara, and K. Cohn-Gordon. 2022. The Messaging Layer Security (MLS) Protocol. IETF Draft. https://datatracker.ietf.org/doc/html/draft-ietf-mls-protocol-15

[16] 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

[17] 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

[18] Gilles Barthe, Thomas Espitau, Benjamin Grégoire, Justin Hsu, Léo Stefanesco, 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

[19] Gilles Barthe, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2016. A Program Logic for Union Bounds. In *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy (LIPIcs, Vol. 55)*, Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 107:1–107:15. https://doi.org/10.4230/LIPIcs.ICALP.2016.107

[20] 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

[21] 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

[22] 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

[23] 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

[24] Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *J. Log. Algebr. Meth. Program.* 84, 1 (2015), 108–123. https://doi.org/10.1016/j.jlamp.2014.02.001

[25] Mihir Bellare, Dennis Hofheinz, and Eike Kiltz. 2015. Subtleties in the Definition of IND-CCA: When and How Should Challenge Decryption Be Disallowed? *J. Cryptol.* 28, 1 (2015), 29–48. https://doi.org/10.1007/s00145-013-9167-4

[26] 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, Vol. 4004)*, Serge Vaudenay (Ed.). Springer, 409–426. https://doi.org/10.1007/11761679_25

[27] 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

[28] Daniel J. Bernstein. 2009. Cryptography in NaCl. https://cr.yp.to/highspeed/naclcrypto-20090310.pdf

[29] 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).

[30] 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

[31] 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

[32] Chris Brzuska, Eric Cornelissen, and Konrad Kohbrok. 2021. Cryptographic Security of the MLS RFC, Draft 11. Cryptology ePrint Archive, Paper 2021/137. https://eprint.iacr.org/2021/137

[33] Chris Brzuska, Antoine Delignat-Lavaud, Christoph Egger, Cédric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. 2021. Key-schedule Security for the TLS 1.3 Standard. Cryptology ePrint Archive, Paper 2021/467. https://eprint.iacr.org/2021/467

[34] 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

[35] Chris Brzuska and Sabine Oechsner. 2021. A State-Separating Proof for Yao's Garbling Scheme. Cryptology ePrint Archive, Paper 2021/1453; To appear at CSF'2023. https://eprint.iacr.org/2021/1453

[36] D. Butler, A. Lochbihler, D. Aspinall, and A. Gascón. 2021. Formalising Σ-Protocols and Commitment Schemes Using CryptHOL. 65, 4 (2021), 521–567. https://doi.org/10.1007/s10817-020-09581-w

[37] Ran Canetti. 2020. Universally Composable Security. *J. ACM* 67, 5 (2020), 28:1–28:94. https://doi.org/10.1145/3402457

[38] 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

[39] Ran Canetti, Alley Stoughton, and Mayank Varia. 2019. EasyUC: Using EasyCrypt to Mechanize Proofs of Universally Composable Security. In *CSF*. IEEE, 167–183.

[40] 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

[41] Lukasz Czajka and Cezary Kaliszyk. 2018. Hammer for Coq: Automation for Dependent Type Theory. *J. Autom. Reason.* 61, 1-4 (2018), 423–453. https://doi.org/10.1007/s10817-018-9458-4

[42] Ivan Damgaard. 2011. On Sigma-Protocols. lecture notes, Aarhus University. https://cs.au.dk/~ivan/Sigma.pdf

[43] François Dupressoir, Konrad Kohbrok, and Sabine Oechsner. 2022. Bringing State-Separating Proofs to EasyCrypt - A Security Proof for Cryptobox. In *To appear in 35th IEEE Computer Security Foundations Symposium*. IEEE. https://eprint.iacr.org/2021/326

[44] 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

[45] 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

[46] Jeremy Gibbons and Ralf Hinze. 2011. Just do it: simple monadic equational reasoning. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*. 2–14. https://doi.org/10.1145/2034773.2034777

[47] 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

[48] Shai Halevi. 2005. A plausible approach to computer-aided cryptographic proofs. *IACR Cryptol. ePrint Arch.* (2005), 181. http://eprint.iacr.org/2005/181

[49] Carmit Hazay and Yehuda Lindell. 2010. *Efficient Secure Two-Party Protocols - Techniques and Constructions*. Springer. https://doi.org/10.1007/978-3-642-14303-8

[50] 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

[51] G M Kelly. 1982. *Basic Concepts of Enriched Category Theory*. 143 pages. Reprint of the 1982 original [Cambridge Univ. Press, Cambridge; MR0651714].

[52] 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

[53] 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

[54] Kevin Liao, Matthew A. Hammer, and Andrew Miller. 2019. ILC: a calculus for composable, computational cryptography. In *PLDI*. ACM, 640–654.

[55] 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

[56] Saunders Mac Lane. 1978. *Categories for the Working Mathematician* (2 ed.). Graduate Texts in Mathematics, Vol. 5. Springer, New York, NY.

[57] Assia Mahboubi and Enrico Tassi. 2021. Mathematical components. Online book. https://math-comp.github.io/mcb/

[58] Kenji Maillard, Cătălin Hriţcu, 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

[59] 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

[60] Ueli Maurer and Renato Renner. 2011. Abstract Cryptography. In *Innovations in Computer Science - ICS'11*. Tsinghua University Press, 1–21. http://conference.iiis.tsinghua.edu.cn/ICS2011/content/papers/14.html

[61] 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

[62] Robin Milner, Mads Tofte, and Robert Harper. 1990. *The Definition of Standard ML*. MIT Press.

[63] 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

[64] 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

[65] Christine Paulin-Mohring, David Baelde, and Pierre Courtieu. 2009. ALEA Coq Library. https://github.com/coq-community/alea

[66] 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, Vol. 9036)*, Riccardo Focardi and Andrew C. Myers (Eds.). Springer, 53–72. https://doi.org/10.1007/978-3-662-46666-7_4

[67] 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

[68] Gordon D. Plotkin and John Power. 2003. Algebraic Operations and Generic Effects. *Appl. Categorical Struct.* 11, 1 (2003), 69–94. https://doi.org/10.1023/A:1023064908962

[69] 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, Vol. 5502)*, Giuseppe Castagna (Ed.). Springer, 80–94. https://doi.org/10.1007/978-3-642-00590-9_7

[70] 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

[71] Eric Rescorla. 2018. The Transport Layer Security (TLS) Protocol Version 1.3. IETF RFC 5246. https://tools.ietf.org/html/rfc8446

[72] Phillip Rogaway and Yusi Zhang. 2018. Simplifying Game-Based Definitions - Indistinguishability up to Correctness and Its Application to Stateful AE. In *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10992)*, Hovav Shacham and Alexandra Boldyreva (Eds.). Springer, 3–32. https://doi.org/10.1007/978-3-319-96881-0_1

[73] 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

[74] Mike Rosulek. 2021. The Joy of Cryptography. Online textbook. http://web.engr.oregonstate.edu/~rosulekm/crypto/

[75] Claus Schnorr. 1991. Efficient signature generation by smart cards. *Journal of Cryptology* 4 (01 1991), 161–174. https://doi.org/10.1007/BF00196725

[76] 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

[77] Nikolaj Sidorenco, Sabine Oechsner, and Bas Spitters. 2021. Formal security analysis of MPC-in-the-head zero-knowledge protocols. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. 1–14. https://doi.org/10.1109/CSF51468.2021.00050

[78] Nikolaj Sidorenco and Bas Spitters. 2022. A formal security analysis of Blockchain voting. (2022).

[79] 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.

[80] Alley Stoughton, François Dupressoir, Pierre-Yves Strub, César Kunz, Juan Manuel Crespo, Benjamin Grégoire, Gilles Barthe, and Benedikt Schmidt. 2018. *EasyCrypt Reference Manual*. Technical Report. 206 pages.

[81] Pierre-Yves Strub. 2020. XHL. https://github.com/strub/xhl

[82] Nikhil Swamy, Cătălin Hriţcu, 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/

[83] Christian Urban. 2008. Nominal Techniques in Isabelle/HOL. *J. Autom. Reason.* 40, 4 (2008), 327–356. https://doi.org/10.1007/s10817-008-9097-2

[84] Philip Wadler. 1990. Comprehending Monads. In *LFP'90* (Nice, France). ACM, New York, NY, USA, 61–78. https://doi.org/10.1145/91556.91592

[85] 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

Table 4. Pkgen code comparison of PKE-CCA^b and AUX^b

PKE-CCA^b.Pkgen()	AUX^b.Pkgen()
	`pk ← get pk_loc`
	`assert pk = ⊥`
`sk ← get sk_loc`	`sk ← get sk_loc`
`assert sk = ⊥`	`assert sk = ⊥`
`(pk, sk) ← η.kgen`	`(pk, sk) ← η.kgen`
`put pk_loc := Some pk`	`put pk_loc := Some pk`
`put sk_loc := Some sk`	`put sk_loc := Some sk`
`return pk`	`return pk`

A PACKAGE EQUIVALENCE FOR KEM-DEM

In order to prove Theorem 6.2, we show that PKE-CCA^b and AUX^b (note that, as in §6, we assume we are given a KEM η and a DEM θ corresponding to the ones used in both games) are perfectly indistinguishable using Theorem 2.4. After inlining package composition we end up with the code comparison found in Table 4, Table 5, and Table 6. We deliberately add extra newlines to align similar lines of code.

Because only AUX^b makes use of the KEY package, the k_loc 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^b, ek_loc will in fact contain the result of the encapsulation of the value stored in k_loc.

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 PKE-CCA^b and AUX^b.

Equivalence for Pkgen. When looking at Table 4, we can see that the only difference is in the first two lines of AUX^b which are absent from 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 ⊥ then pk_loc does too. To exploit the invariant, we first swap commands on the right-hand side to perform the read of sk on both sides and we recover the fact that it must be ⊥. 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 5 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 one-sided rules for memory reads and writes that *remember* values that have been written and read; they correspond to rules like get-lhs or put-lhs that are presented at the end of §4.1. With this we have the same programs on both sides up to the following line:

```
(k, ek) ← η.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 η (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

Table 5. Pkenc code comparison of $\mathsf{PKE\text{-}CCA}^b$ and AUX^b

$\mathsf{PKE\text{-}CCA}^b$.Pkenc(msg)	AUX^b.Pkenc(msg)
pk ← get pk_loc	pk ← get pk_loc
assert pk ≠ ⊥ as pkSome	assert pk ≠ ⊥
let pk := getSome pk pkSome in	
ek ← get ek_loc	ek ← get ek_loc
assert ek = ⊥	assert ek = ⊥
c ← get c_loc	c ← get c_loc
assert c = ⊥	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)	(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	if b then
c ← θ.enc(k, msg)	c ← θ.enc(k,msg)
else	else
c ← θ.enc(k, 0)	c ← θ.enc(k,0)
put ek_loc := Some ek	put c_loc := Some c
put c_loc := Some c	put c_loc := Some c
return (ek, c)	return (ek, c)

the right-hand side we make use of the invariant relating pk, k_loc, and ek_loc to ascertain that since ek_loc contains ⊥, 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 6, the equivalence proof is conducted in roughly the same way as above. Then we proceed with a case-analysis on ek = Some ek'. In the else branch, all of the asserts hold, as they hold in the lines above and the invariant stating that pk_loc is ⊥ if and only if sk_loc is satisfied, and furthermore the case-analysis yielded

Table 6. Pkdec code comparison of PKE-CCA^b and AUX^b

PKE-CCA^b.Pkdec(ek',c')	AUX^b.Pkdec(ek',c')
sk ← get sk_loc	pk ← get pk_loc
assert sk ≠ ⊥ as skSome	assert pk ≠ ⊥
let sk := getSome sk skSome in	
ek ← get ek_loc	ek ← get ek_loc
c ← get c_loc	c ← get c_loc
assert (	assert (
(ek, c) ≠ (Some ek', Some c')	(ek, c) ≠ (Some ek', Some c')
)	)
	if ek = Some ek' then
	c ← get c_loc
	assert c ≠ Some 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 ≠ Some ek'
	k' ← η.decap(sk,ek')
k ← η.decap(sk, ek')	msg ← θ.dec(k', c')
return θ.dec(k, c')	return msg

ek ≠ Some 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 η.decap of the KEM. Instead, we exploit the invariant that states that the stored encrypted key in ek_loc corresponds to the encryption of the key in k_loc using the public key in pk_loc, a fact which we encoded by saying that in this case k is equal to η.encap(sk, ek). We conclude remembering that we are in the branch where ek = Some ek'.

Now that the three procedures have been shown equivalent, we know that the two packages are indeed perfectly indistinguishable.