SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq

Carmine Abate[1] Philipp G. Haselwarter[2] Exequiel Rivas[3] Antoine Van Muylder[4] Théo Winterhalter[1]
Cătălin Hriţcu[1] Kenji Maillard[5] Bas Spitters[2]

[1]MPI-SP [2]Aarhus University [3]Inria Paris [4]Vrije Universiteit Brussel [5]Inria Rennes

***Abstract*—State-separating proofs (SSP) is a recent methodology for structuring game-based cryptographic proofs in a modular way. While very promising, this methodology was previously not fully formalized and came with little tool support. We address this by introducing SSProve, the first general verification framework for machine-checked state-separating proofs. SSProve combines high-level modular proofs about composed protocols, as proposed in SSP, with a probabilistic relational program logic for formalizing the lower-level details, which together enable constructing fully machine-checked crypto 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.**

1 Introduction

Cryptographic proofs can be challenging to make fully precise and to rigorously check. This has caused a "crisis of rigor" [16] in cryptography that Shoup [49], Bellare and Rogaway [16], Halevi [31], 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 [8, 43] and Isabelle/HOL [15]. It has also led to the development of specialized proof assistants [12] and automated verification tools for crypto proofs [11, 14, 21]. There are two key ideas behind these tools: (i) formally representing games and the adversaries against them as code in a probabilistic programming language, and (ii) using program verification techniques to conduct all game transformation steps in a machine-checked manner.

For a long time however, game-based proofs have lacked modularity, which made them hard to scale to large, composed protocols such as TLS [46] or the upcoming MLS [7]. To address this issue, Brzuska et al. [22] have recently introduced *state-separating proofs (SSP)*, a methodology for modular game-based proofs, inspired by the paper proofs in the miTLS project [19, 20, 29], by prior compositional cryptography frameworks [23, 39], and by process algebras [40]. In the SSP methodology, the code of cryptographic games is split into packages, which are modules made up of procedures sharing state. Packages can call each other's procedures (also known as oracles) and can operate on their own state, but cannot directly access other packages' state. Packages have natural notions of sequential and parallel composition that satisfy simple algebraic laws, such as associativity of sequential composition. This law is used to define cryptographic reductions not only in SSP, but also in the *The Joy of Cryptography* textbook [48], which teaches crypto proofs in a style very similar to SSP.

While the SSP methodology is very promising, the lack of a complete formalization makes it currently only usable for informal paper proofs, not for machine-checked ones. The SSP paper [22] defines package composition and the syntax of a cryptographic pseudocode language for games and adversaries, but the semantics of this language is not formally defined, and the meaning of their `assert` operator is not even clear, 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 crypto 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 [8, 31, 43, 52], or does not exactly match the SSP package abstraction [12, 35] (see §6 for details).

The main contribution of this work is to introduce SSProve, the first general verification framework for machine-checked state-separating proofs. SSProve brings together two different proof styles into a single unified framework: (1) high-level proofs are modular, done by reasoning equationally about composed packages, as proposed in SSP [22]; (2) low-level details are formally proved in a probabilistic relational program logic [8, 12, 43]. Importantly, we show a formal connection between these two proof styles.

SSProve is, moreover, a foundational framework, fully formalized itself in Coq. For this we define the syntax of crypto pseudocode in terms of a free monad, in which external calls are represented as algebraic operations [44]. This gives us a principled way to define sequential composition of packages based on an algebraic effect handler [45] and to give machine-checked proofs of the SSP package laws [22], some which were treated informally on paper. We moreover make precise the minimal state-separation requirements between adversaries and the games with which they are composed—this reduces proof burden and allows us to prove more meaningful security results, that do not require the adversary's state to be disjoint from intermediate games in the proof.

Beyond just syntax, we also give a denotational semantics to crypto 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 relating pairs of crypto 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. [37], 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 simply applying a relative monad transformer. Working out this instance of Maillard et al.'s [37] recipe involved formalizing various non-standard categorical constructs in Coq, in an order-enriched context: lax functors, lax natural transformations, left relative adjunctions, lax morphisms between such adjunctions, state transformations of such adjunctions, etc. This formalization is of independent interest and should also allow us to more easily add extra side-effects and F*-style sub-effecting [52] to SSProve in the future.

Outline. The remainder of this paper is structured as follows. §2 illustrates the key ideas of how to use SSProve on two simple crypto proofs, showing semantic security of ElGamal and PRF-based encryption. In §3 we formalize the SSP methodology: cryptographic pseudocode, packages, sequential and parallel composition, and the algebraic laws they satisfy. In §4 we introduce the rules of our probabilistic relational program logic and use them to prove Theorem 1, 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. Finally, §6 discusses related work and §7 future directions.

The full formalization of SSProve and of the examples from this paper (close to 20K lines of Coq code including comments) are available under the MIT open source license at `https://github.com/SSProve/ssprove`

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. [22]. The main concept behind this

```
package: P
mem: n : nat
---------------------------------
X(b):                 Y():
  if b then             return 5
    return 1
  else
    return Z(n)
```

Figure 2. Possible pseudocode implementation for P.

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.

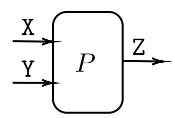

Figure 1. Package P.

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. 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 $P.\texttt{X}$.

Package algebra. Packages can be combined as algebraic objects. We can build complex packages out of simpler ones using the following composition operations.[2]

- *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 two packages 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 provides the procedures from both P_1 and P_2.
- *Identity package*: given an interface I, we have a package that simply forwards all calls in this interface. We refer to it as the identity package on the interface I, written $\texttt{ID}_I$, and we have that $\texttt{import}(\texttt{ID}_I) = \texttt{export}(\texttt{ID}_I) = I$.

These operations have graphical counterparts which we show in Figure 3. Moreover, there are natural algebraic laws that hold between these operators. For example, sequential

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

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

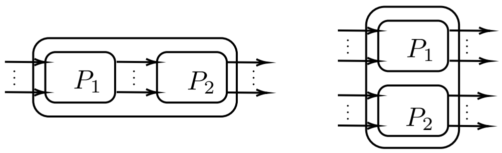

Figure 3. Sequential and parallel composition.

composition is an associative operator. Such laws are convenient for cryptographic proofs, since they allow the compositional structure of a package to be manipulated without having to look at all at the implementation of its procedures.

Games and distinguishers. A package with no imports is called a *game*. A game pair contains two games that export the same procedures, i.e. a tuple (G^0, G^1) such that $\texttt{export}(G^0) = \texttt{export}(G^1)$ and $\texttt{import}(G^0) = \texttt{import}(G^1) = \emptyset$. 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 (so without imports) exports a single procedure $\texttt{run} : \texttt{unit} \to \texttt{bool}$ as above, we denote by $\Pr[b \leftarrow G]$ the probability that $G.\mathsf{run}$ returns the boolean value b when running on initial memory. We can quantify how much a distinguisher can distinguish the two packages in a game pair:

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

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

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

Lemma 1 (Triangle Inequality)**.** *Let G^0, G^1 and G^2 be games, we have that for every distinguisher $\mathcal{D}$,*

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

Proof. By unfolding Definition 1 we have

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

□

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

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

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

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

□

As its name indicates, Lemma 2 is used to reduce the advantage of the distinguisher over a composed game ($M{\circ}G^b$), to the advantage over part of the game (M), for which we know a bound. We will use both these SSP lemmas in §2.3.

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

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

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

2.2 Proving perfect indistinguishability steps in a probabilistic relational program logic

We now present a 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. [8]. Judgments of this logic are of the form

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

and convey the intuition 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. In this paper we use the $p.M$ convention to denote a function that binds p and has body M (usually denoted by $\lambda p.M$ in the functional programming community). This notation will be handy for writing postconditions, which depend on final memories and on final results. We now state the main theorem of SSProve:

Theorem 1. *Let $G = (G^0, G^1)$ be a game pair with respect to export interface $\mathcal{E} = \mathtt{export}(G^0) = \mathtt{export}(G^1)$. 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 $a \in A$,

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

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

Intuitively, we ask that both procedures, run on memories satisfying 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. 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 crypto proof by Brzuska et al. [22] that we have verified in Coq using our framework. In this proof, reasoning about composed packages (using Lemmas 1 and 2 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 [22] these steps were only justified *informally* by code inspection. Instead, we have *formally* verified these steps too, using Theorem 1 and our relational program logic.

Brzuska et al. [22] 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 to the security of the PRF, expressed as being *indistinguishable* from a package doing random sampling.

The scheme assumes a 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 as the probability for an adversary to distinguish it from a package that samples from an uniform distribution (*real vs random* paradigm [48]). Formally, given the packages $\mathtt{PRF}^0$ and $\mathtt{PRF}^1$ as in Figure 4, the security of PRF, $\alpha(\mathtt{PRF})(\mathcal{A})$ is defined using Definition 1 as the advantage of an adversary for the game pair $\mathtt{PRF} = (\mathtt{PRF}^0, \mathtt{PRF}^1)$:

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

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

```
package: PRF⁰
mem: k : option KEY

EVAL(x):
 if k = ⊥ then
   k <$ uniform {0,1}^n ;;
 return prf(k, x)
```

```
package: PRF¹
mem: T : map [KEY -> KEY]

EVAL(x):
 if T[x] = ⊥ then
   T[x] <$ uniform{0,1}^n ;;
 return T[x]
```

Figure 4. Packages $\mathtt{PRF}^0$ and $\mathtt{PRF}^1$.

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

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

dec(k, (r, c)):
  pad ← prf(k, r) ;;
  m ← c xor pad ;;
  return m
```

Figure 5. Algorithms for `prf`-based encryption scheme.

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 $(\mathtt{IND\text{-}CPA}^0, \mathtt{IND\text{-}CPA}^1)$, and the packages involved are introduced in Figure 6. Notice that in procedure $\mathtt{IND\text{-}CPA}^1$`.ENC` the argument `m` is never used, the encryption procedure is run on a random `m'`. Therefore the advantage of an adversary w.r.t. the game $(\mathtt{IND\text{-}CPA}^0, \mathtt{IND\text{-}CPA}^1)$ represents the probability that the adversary is able to distinguish the encryption of `m` from the encryption of a random bit-string. The security of the encryption procedure with respect to an adversary $\mathcal{A}$ is then $\alpha(\mathtt{IND\text{-}CPA})(\mathcal{A})$.

Brzuska et al. [22] use a sequence of *game-hops* to bound $\alpha(\mathtt{IND\text{-}CPA})$ in terms of (a linear function of) $\alpha(\mathtt{PRF})$. This technique of game-hops follows the style of equational-like reasoning chains from §2.1 (Lemma 1 in particular), 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, $\mathtt{IND\text{-}CPA}^b$ is shown equivalent to a variant, $\mathtt{MOD\text{-}CPA}^b$, that gets the secret key through the PRF, i.e., with a call to `EVAL` of the package $\mathtt{PRF}^0$ or $\mathtt{PRF}^1$ (see Figure 7). By repeatedly applying Lemma 1, we bound $\alpha(\mathtt{IND\text{-}CPA})(\mathcal{A})$

```
package: IND-CPA⁰
mem: k : option KEY

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

```
package: IND-CPA¹
mem: k : option KEY

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

Figure 6. Packages $\mathtt{IND\text{-}CPA}^0$ and $\mathtt{IND\text{-}CPA}^1$.

```
package: MOD-CPA^0
mem:

ENC(m):
 r <$ uniform {0,1}^n ;;
 pad ← EVAL(r) ;;
 c ← m xor pad ;;
 return (r, c)
```

```
package: MOD-CPA^1
mem:

ENC(m):
 m' <$ uniform {0,1}^n ;;
 r <$ uniform {0,1}^n ;;
 pad ← EVAL(r) ;;
 c ← m' xor pad ;;
 return (r, c)
```

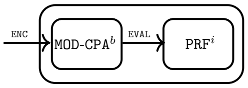

Figure 7. Packages $\texttt{MOD-CPA}^b$ import $\texttt{EVAL}$ from $\texttt{PRF}^i$

$\texttt{IND-CPA}^0\texttt{.ENC(m)}$

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

$(\texttt{MOD-CPA}^0 \circ \texttt{PRF}^0)\texttt{.ENC(m)}$

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

Figure 8. ENC procedures expanded

by

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

By observing that $\alpha(\texttt{IND-CPA}^0, \texttt{MOD-CPA}^0 \circ \texttt{PRF}^0)(\mathcal{A}) = 0$, and $\alpha(\texttt{MOD-CPA}^1 \circ \texttt{PRF}^0, \texttt{IND-CPA}^1)(\mathcal{A}) = 0$, and by using Lemma 2 twice, we reduce this bound to

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

where $\varepsilon_{stat.}{=}\alpha(\texttt{MOD-CPA}^0 \circ \texttt{PRF}^1,\ \texttt{MOD-CPA}^1 \circ \texttt{PRF}^1)$. The advantage of an attacker w.r.t $\texttt{MOD-CPA}^0$ and $\texttt{MOD-CPA}^1$ is usually referred to as *statistical gap*—a polynomial function of the number of calls from the adversary (see [22, appendix A]).

It remains to justify the two perfect indistinguishabilities stated above. These steps involve replacing an informal argument [22] by a fully formal one, moving to our probabilistic relational program logic, as such we will detail one of them: $\alpha(\texttt{IND-CPA}^0, \texttt{MOD-CPA}^0 \circ \texttt{PRF}^0)(\mathcal{A}) = 0$. The other one $\alpha(\texttt{MOD-CPA}^1 \circ \texttt{PRF}^0, \texttt{IND-CPA}^1)(\mathcal{A}) = 0$ is analogous.

In order to prove this equivalence, Brzuska et al. [22] notice that the ENC procedures of $\texttt{IND-CPA}^0$ and $\texttt{MOD-CPA}^0 \circ \texttt{PRF}^0$ (see Figure 8) return the same ciphertext when called on the same m. The two procedures are obtained by "inlining" the code of $\texttt{PRF}^0$.EVAL inside $\texttt{MOD-CPA}^0$, and by "unfolding" the code of enc.

The only difference between the left and the right side is in the case `k = ⊥` and w.r.t. `k <$ uniform {0,1}^n` that on the left is the first command to be executed and on the right only comes after `r <$ uniform {0,1}^n`, another *independent* random sampling. Here Brzuska et al. [22] conclude informally that independence allows to "swap" the two operations. We instead use Theorem 1 to formally reduce

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

```
Enc(A, M):
 b <$ uniform{0,..n-1} ;;
 B ← g^b ;;
 return (B, M * A^b)

Dec(a, (B, X)):
 return X * (B^a)^-1
```

Figure 9. Algorithms for ElGamal encryption scheme.

$\alpha(\texttt{IND-CPA}^0, \texttt{MOD-CPA}^0 \circ \texttt{PRF}^0)(\mathcal{A}){=}0$ 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 m,

$$\vDash \{m_0 = m_1\}\ \texttt{IND-CPA}^0\texttt{.ENC(m)} \sim$$
$$\left(\texttt{MOD-CPA}^0 \circ \texttt{PRF}^0\right)\texttt{.ENC(m)}\{(c_0, c_1).m_0 = m_1 \wedge c_0 = c_1\}.$$

This judgment intuitively states that encrypting m with the same initial memories "$m_0 = m_1$", terminates still in memories and ciphertexts drawn from the same distribution, "$m_0 = m_1 \wedge c_0 = c_1$". We use the following instance of the swap rule from Figure 13, to formally justify this swapping:

$$\frac{\begin{array}{c}\vDash \{m_0 = m_1\}\ \texttt{k <\$ uniform\{0,1\}}^{\texttt{n}} \sim \texttt{r <\$ uniform\{0,1\}}^{\texttt{n}}\ \{m_0 = m_1 \wedge c_0 = c_1\} \\ \vDash \{m_0 = m_1\}\ \texttt{r <\$ uniform\{0,1\}}^{\texttt{n}} \sim \texttt{k <\$ uniform\{0,1\}}^{\texttt{n}}\ \{m_0 = m_1 \wedge c_0 = c_1\}\end{array}}{\begin{array}{c}\vDash \{m_0 = m_1\} \\ \texttt{k <\$ uniform \{0,1\}}^{\texttt{n}}\texttt{ ;; r <\$ uniform \{0,1\}}^{\texttt{n}} \sim \\ \texttt{r <\$ uniform \{0,1\}}^{\texttt{n}}\texttt{ ;; k <\$ uniform \{0,1\}}^{\texttt{n}} \\ \{m_0 = m_1 \wedge c_0 = c_1\}\end{array}}$$

2.4 Security proof of ElGamal in SSProve

We also illustrate the key ideas of SSProve on a security proof for the ElGamal public-key encryption scheme inspired by *The Joy of Cryptography* textbook [48, Chapter 15.3]. ElGamal is parameterized by a multiplicative cyclic group $(\mathcal{G}, *)$ with n elements and generated by g, usually denoted by $\langle g \rangle = \mathcal{G}$. Plaintexts are elements $M \in \mathcal{G}$ and ciphertexts are pairs of group elements $C = (C_1, C_2) \in \mathcal{G} \times \mathcal{G}$. Secret keys are elements of $\mathbb{Z}_n$, while public keys are group elements once again, $A \in \mathcal{G}$. The key generation algorithm (KeyGen in Figure 9) generates a secret key that is a random number $a \in \{0, \dots, n-1\}$ and a public key that is g^a. Encryption and decryption (Enc and Dec Figure 9) involve the group operation (`_*_`), exponentiation (`_)`- and the multiplicative inverse (`_)^-1`.

Under the *Decisional Diffie–Hellman* (DDH) assumption for the group $\mathcal{G}$, i.e. $\texttt{DDH}^0$ and $\texttt{DDH}^1$ 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 restricts to 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 [48, Claim 15.5]. We leave the formalization of this last result as future work and discuss hereafter our proof of OT-CPA in SSProve.

OT-CPA is expressed in terms of the advantage against game pair $(\texttt{CPA}^0, \texttt{CPA}^1)$ in Figure 11. Both packages return

```
package: DDH^0
mem:
pk : option PUBKEY
a : option SECKEY

QUERY()
 a <$ uniform {0,..n-1};;
 b <$ uniform {0,..n-1};;
 pk ← g^a ;;
 return (g^a,g^b,g^ab)
```

```
package: DDH^1
mem:
pk : option PUBKEY
a : option SECKEY

QUERY()
 a <$ uniform {0,..n-1};;
 b <$ uniform {0,..n-1};;
 c <$ uniform {0,..n-1};;
 pk ← g^a ;;
 return (g^a,g^b,g^c)
```

Figure 10. The DDH assumption states that $\mathtt{DDH}^0$ and $\mathtt{DDH}^1$ are computationally indistinguishable.

```
package: CPA^0
mem:
pk : option PUBKEY
sk : option SECKEY
counter : nat

ENC(M):
 if counter = 0 then
  (pk, sk) ← KeyGen() ;;
  (B, C) ← Enc (pk, M) ;;
  counter++ ;;
  return (B, C)
 else
  return ⊥
```

```
package: CPA^1
mem:
pk : option PUBKEY
sk : option SECKEY
counter : nat

ENC(M):
 if counter = 0 then
  (pk, sk) ← KeyGen() ;;
  (B, C) <$ uniform GxG ;;
  counter++ ;;
  return (B, C)
 else
  return ⊥
```

Figure 11. Packages $\mathtt{CPA}^0$ and $\mathtt{CPA}^1$ in ElGamal.

a ciphertext only if the counter is 0, so the adversary can only see one ciphertext. Both packages call `KeyGen` to generate public and private keys, but while $\mathtt{CPA}^0$ effectively encrypts the message provided by the adversary with the public key through `Enc (pk, M)`, $\mathtt{CPA}^1$ returns a randomly sampled ciphertext `(B,C) <$ uniform GxG`, i.e. a pair of group elements sampled uniformly at random from $\mathcal{G} \times \mathcal{G}$.

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

$$\alpha(\mathtt{CPA})(\mathcal{A}) \leq \alpha(\mathtt{DDH})(\mathcal{A} \circ \mathtt{Aux}).$$

We once again obtain this result by first repeatedly applying

```
package: Aux
mem:  counter : nat

ENC(M):
 if counter = 0 then
  (A, B, C) ← QUERY() ;;
  counter++ ;;
  return (B, M * C)
 else
  return ⊥
```

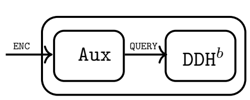

Figure 12. Package `Aux` imports `QUERY` from $\mathtt{DDH}^b$.

Lemma 1 to bound $\alpha(\mathtt{CPA})(\mathcal{A})$ by

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

We will see that the first and last advantages are null by proving the packages perfectly indistinguishable, and the remaining advantage is equal to $\alpha(\mathtt{DDH})(\mathcal{A} \circ \mathtt{Aux})$ by simple application of Lemma 2. It now remains to show the equivalences below:

Step $\alpha(\mathtt{CPA}^0, \mathtt{Aux} \circ \mathtt{DDH}^0)(\mathcal{A}){=}0$: We apply Theorem 1 and reduce the goal to a relational judgment between $\mathtt{CPA}^0$`.ENC(M)` and $(\mathtt{Aux} \circ \mathtt{DDH}^0)$`.ENC(M)` for a generic plaintext `M`, and where the invariant ψ is equality of memories. Inlining the code of `QUERY` provided by $\mathtt{DDH}^0$ inside `Aux` and unfolding one realizes the two code fragments coincide and the judgment holds by application of the `reflexivity` rule in Figure 13.

Step $\alpha(\mathtt{Aux} \circ \mathtt{DDH}^1, \mathtt{CPA}^1)(\mathcal{A}){=}0$: This step is quite similar to the one above. After inlining however the two code fragments are not exactly the same, since in particular $\mathtt{CPA}^1$ completely ignores `M` and returns a random ciphertext, while $\mathtt{Aux} \circ \mathtt{DDH}^1$ returns $\mathtt{M{*}g}^{\mathtt{c}}$ for a random `c`. 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 [8, 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).

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 [41, 53]. 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 [43] of embedding the pure fragment of our cryptographic programming language shallowly in Coq, and embedding the effects deeply via a free monad. This free monad is defined as an inductive type:

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

This type of raw code comes equipped with an induction principle, which is used for instance in the proof of Theorem 1, in Theorem 4.2, and in the definition of the bind operation and sequential composition of packages by recursion over code.

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

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

```
get ℓ (λ x_ℓ . put ℓ (x_ℓ + 1) (ret x_ℓ))                          (1)
sample (uniform {0,1}^n)
       (λ y . call prf (y, 101010) (λ z . ret z))                   (2)
```

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

Valid code Raw code is merely a representation of syntax. To record which probabilistic sampling operations, imported procedures, and locations are used, we introduce a notion of valid code. Validity is defined relatively to a collection of sampling operations `Op`, a set of locations $\mathcal{L}$, and finally an import *interface* $\mathcal{I}$ which is a set of procedure signatures (`opsig`) consisting of a name, an input type and an output type. Concretely, the code in (1) is valid with respect to $\{\ell : \texttt{nat}\}$ and the empty import interface, while (2) is valid with respect to the empty set of locations and the interface $\{\texttt{prf} : \texttt{nat} \times \texttt{nat} \to \texttt{nat}\}$, assuming further that `uniform` $\{0,1\}^n$ `: Op` is a valid sampling operation. The type `code` is then simply defined as valid raw code, i.e. $\texttt{code}_{\mathcal{L},I}\ \texttt{A} = \{\ \texttt{c} : \texttt{raw\_code A} \mid \texttt{is\_valid c}\ \mathcal{L}\ \mathcal{I}\ \}$, where in the paper we sometimes omit the set of locations and the interface. Thanks to the use of tactics and Coq's type classes, proofs of validity for well-scoped user-written code are constructed automatically without requiring user intervention.

Continuations A *continuation* is a suspended computation awaiting the result of an operation. Consider for instance the code (1). The `get` operation performs a memory lookup at the location ℓ, and its continuation is a Coq function (λ x_ℓ . put ...) of type (`type` $\ell \to$ `raw_code nat`) which 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 commonly written as ℓ++.

Variables As demonstrated in example (1), 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 (1), this value is available in the continuation of `get` ℓ (λ x_ℓ . put ...) as x_ℓ. Formally speaking, x_ℓ is an immutable Coq variable, and in (1) the location ℓ itself is a Coq variable of type `Location`.

Monadic bind The `bind` operation of the monad, with type `code A` $\to$ (`A` $\to$ `code B`) $\to$ `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 ret m₁ else ret m₂)
Definition κ : nat → code nat := λm. put ℓ m (ret 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
  | ret a ⇒ κ a
  | call p x κ' ⇒ call p x (λ p . bind (κ' p) 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.

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

```
Fixpoint for_loop (N : nat)
                  (c : nat → code unit) : code unit :=
  match N with
  | 0 ⇒ c 0
  | S m ⇒ bind (for_loop m 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 ⇒ ret false
  | S n ⇒ bind c (fun b ⇒ match b with
                          | false ⇒ ret true
                          | true  ⇒ do_while n c
                          end)
  end.
```

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

Standard subdistributions Probabilistic operations denoting a collection of subdistributions we may sample from are included in the parameter type `Op`. Standard subdistributions including uniform sampling on finite types as well as a null subdistribution are predefined for convenience. The null subdistribution in particular allows us to represent `assert`.

```
Definition assert (b : bool) : code unit :=
    if b then ret tt else sample null (λ F . ret F)
```

Here `unit` stands for the Coq singleton type with a unique inhabitant `tt`. If `b` is `true`, then `assert` returns the trivial value `tt`, but if `b` is `false`, we instead sample from the `null` distribution, which assigns probability zero to the `tt` values. Sampling from the null subdistribution is similar to non-termination, and it means that the continuation will never be called.

Procedure calls A call to an external procedure such as `prf` in (2) is represented by the `call` operation, taking a procedure name `p` annotated with type, a value matching the argument type of `p`, and a continuation matching the return type of `p`. In §3.3 we show how an implementation gets substituted for this placeholder via sequential packages composition.

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

```
ret v                  :=   ret v
x ← c₁ ;; c₂           :=   bind c₁ (λx. c₂)
x ← call(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)
```

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 (1) 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 ;;
                    ret y
             | Some val_k' ⇒  ret val_k'
             end) ;;
  val_prf ← call(prf, (val_k, x)) ;;
  ret val_prf.
```

Here we freely mix constructors of `raw_code` with other Gallina terms such as the `match _ with _ ⇒ _ end` construct. The result of the `match` is made available to the continuation of the code as `val_k` via a use of `bind`.

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` $\to$ `SD (A` $\times$ `mem)`

This semantics is similar to that of Barthe et al. [9]. 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` [1, 36], a Coq library for real analysis. The semantics function `Pr_code` is defined by recursion on the structure of `c`. Its definition basically boils down to providing an effect handler that interprets state and probabilities in the monad `mem` $\to$ `SD(`$-$ $\times$ `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 $\Pr[b \leftarrow G] = \mathtt{d}(b)$ as the probability assigned to b by this subdistribution on booleans.

3.3 Packages

A raw package is a finite map from names to raw procedures. An *interface* is a finite set of operation signatures (`opsig`), each specifying the name, argument type, and result type of a procedure. A *package* is then a raw package *RP* together with an import interface $\mathcal{I}$, an export interface $\mathcal{E}$, and a set of locations $\mathcal{L}$, such that each procedure in *RP* is valid with respect to $\mathcal{L}$ and $\mathcal{I}$, and each procedure name listed in $\mathcal{E}$ is implemented by a procedure in *RP* of the appropriate type.

```
package: ℒ
mem:  counter : nat
---------------------------
ENC(M):
 if counter = 0 then
  (A,B,C) ← QUERY();;
  counter++ ;;
  ret (B, M * C)
 else
  ret ⊥
```

Consider for instance the package $\mathcal{L}$ from Figure 12. The memory used, `mem`$(\mathcal{L})$, consists of one location {`counter : nat`}. the import interface `import`$(\mathcal{L})$ contains a single procedure {`QUERY :` `unit` $\to \mathcal{G} \times \mathcal{G} \times \mathcal{G}$}. There is one procedure implemented by $\mathcal{L}$, yielding an export interface `export`$(\mathcal{L}) =$ {`ENC :` $\mathcal{G} \to$ `option` $(\mathcal{G} \times \mathcal{G})$}.

We define composition of packages, following Brzuska et al. [22]. Given two raw packages P, Q we may define their *sequential composition* $Q \circ P$ by traversing Q and replacing each `call` by the corresponding procedure implementation in P. In case P does not implement the searched for procedure, we use a default value instead. If additionally, the exports of P match the imports of Q, i.e. `import`$(Q) \subseteq$ `export`(P), and both packages are valid, then so is $Q \circ P$, in which case no default value is needed. We have `mem`$(Q \circ P) =$ `mem`$(P) \cup$ `mem`(Q), `import`$(Q \circ P) =$ `import`(P) and `export`$(Q \circ P) =$ `export`(Q). Concretely, during the traversal each `call p a` κ node is replaced by bind (P.p a) (λ x . $\texttt{link}_P$ (κ x)) where $\texttt{link}_P$ stands for composition with P on the remaining code. Experts will recognize this transformation as an algebraic effect handler, interpreting the free monad for probabilities, state and the operations imported by P to code in the free monad for probabilities, state, and the operations imported by Q.

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

Private state When formalizing composition in SSProve we do not impose restrictions on the disjointness of the state that P and Q manipulate. The two lemmas from §2.1 and the SSP package laws below hold without any such assumptions. The essence of state separation can be thus viewed as disjointness of state between the adversary and the games in a pair. We thus introduce the more economical assumption that only *the adversary* has to have disjoint state in our security definitions (e.g., perfect indistinguishability from §2.1) and corresponding theorem statements (e.g., Theorem 1).

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

3.4 Package laws

We formally proved the algebraic laws obeyed by packages as stipulated by Brzuska et al. [22]. Sequential composition is associative and parallel composition is commutative and associative, so for any packages P_1, P_2, P_3:

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

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

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

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

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

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

As we have hinted before, these laws do not require disjointness of state, because they are pretty syntactic equalities. In fact, in SSProve they hold with respect to the syntactic equality of Coq, without the need to define a separate notion of "code equality" [22].

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 1, the link between the high-level reasoning based on the package laws to the low-level one based on our probabilistic relational program logic in §4.2.

4.1 Selected rules

Our logic exposes relational judgments of the form $\vDash \{pre\}\ c_0 \sim c_1\ \{post\}$, for which a basic intuition is provided in §2.2. Formally, c_0 and c_1 denote probabilistic stateful code with return type A_0 and A_1 respectively, $m_0 : \mathsf{mem}, m_1 : \mathsf{mem} \vdash pre : \mathbb{P}$ is a proposition with free variables m_0 and m_1 denoting the initial state of the memory (before execution of the code), and $m_0 : \mathsf{mem}, m_1 : \mathsf{mem} \vdash post : A_0 \times A_1 \to \mathbb{P}$ is a propositional function on the returned values by the executed code and which also has access to free variables m_0 and m_1 representing the final state of the memory (after execution). The code fragments appearing in a judgment are drawn from the free monad $\texttt{code}_{\mathcal{L},I}$ 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 *pre* is defined to be a relation between initial memories (for instance, $m_0 = m_1$). Similarly the postcondition *post* relates final memories and final results, intuitively obtained after the execution of c_i on m_i. We describe how to assign a formal semantics for such probabilistic judgments in §5.2. The semantics is based on the notion of *probabilistic couplings*, already adopted by Barthe et al. [13]. A coupling of two subdistributions $d_0 : \mathrm{SD}(A_0)$ and $d_1 : \mathrm{SD}(A_1)$, denoted by $d \sim^{d_0}_{d_1}$, is a subdistribution over $A_0 \times A_1$ such that its left and right marginals correspond to d_0 and d_1 respectively. In that respect our judgment $\vDash \{pre\}\ c_0 \sim c_1\ \{post\}$ is valid when there exists a (state-transformed) coupling of the c_i's (seen as state-transformed subdistributions, §3.2) such that *pre* and *post* hold with non zero probability. In the remaining of this subsection we describe a selection of our rules, displayed in Figure 13.

The rule reflexivity relates every piece of code to itself, this is because the execution of the same code on the same initial memories ($m_0 = m_1$) leads to final memories and final results drawn from the same distribution ($m_0 = m_1 \wedge a_0 = a_1$).

$$\frac{c : \texttt{code}\ L\ A}{\vDash \{m_0 = m_1\}\ c \sim c\ \{(r_0, r_1).\ m_0 = m_1 \wedge r_0 = r_1\}}\ \texttt{reflexivity}$$

$$\frac{\begin{array}{c} c_0 : \texttt{code}\ L_0\ A_0 \quad c_1 : \texttt{code}\ L_1\ A_1 \\ f_0 : A_0 \to\ \texttt{code}\ L_0\ B_0 \quad f_1 : A_1 \to\ \texttt{code}\ L_1\ B_1 \\ \vDash \{pre\}\ c_0 \sim c_1\ \{\mu\} \\ \forall a_0 a_1.\ \vDash \{\lambda(h_0, h_1).\ \mu(a_0, h_0)(a_1, h_1)\}\ (f_0\, a_0) \sim (f_1\, a_1)\ \{post\} \end{array}}{\vDash \{pre\}\ a_0 \leftarrow c_0;;\, f_0\, a_0 \sim a_1 \leftarrow c_1;;\, f_1\, a_1\ \{post\}}\ \texttt{seq}$$

$$\frac{\begin{array}{c} c_0 : \texttt{code}\ L\ A_0 \quad c_1 : \texttt{code}\ L\ A_1 \\ \vDash \{I\}\ c_0 \sim c_1\ \{(a_0, a_1).\ I \wedge post(a_0, a_1)\} \\ \vDash \{I\}\ c_1 \sim c_0\ \{(a_1, a_0).\ I \wedge post(a_0, a_1)\} \end{array}}{\vDash \{I\}\ c_0;;\, c_1 \sim c_1;;\, c_0\ \{(a_0, a_1).\ I \wedge post(a_0, a_1)\}}\ \texttt{swap}$$

$$\frac{\begin{array}{c} c_0\ c_0' : \texttt{code}\ L\ A_0 \quad c_1 : \texttt{code}\ J\ A_1 \\ \vDash \{pre\}\ c_0 \sim c_1\ \{post\} \qquad \forall h.\ \theta(c_0\ h) = \theta(c_0'\ h) \end{array}}{\vDash \{pre\}\ c_0' \sim c_1\ \{post\}}\ \texttt{eqDistrL}$$

$$\frac{\begin{array}{c} c_0 : \texttt{code}\ L\ A_0 \quad c_1 : \texttt{code}\ L\ A_1 \\ \vDash \{pre\}\ c_0 \sim c_1\ \{post\} \end{array}}{\vDash \{pre^{-1}\}\ c_1 \sim c_0\ \{post^{-1}\}}\ \texttt{symmetry}$$

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

$$\frac{\begin{array}{c} c_0, c_1 : \texttt{code}\ L\ \texttt{bool} \quad N : \mathbb{N} \\ \vDash \{I(\texttt{true}, \texttt{true})\}\ c_0 \sim c_1\ \{(b_0, b_1).\ b_0 = b_1 \wedge I(b_0, b_1)\} \end{array}}{\begin{array}{c} \vDash \{I(\texttt{true}, \texttt{true})\}\ \texttt{do\_while}\, N\, c_0 \sim \\ \texttt{do\_while}\, N\, c_1\ \{(b_0, b_1).\ b_0 = b_1 = \texttt{false} \vee I(\texttt{false}, \texttt{false})\} \end{array}}\ \texttt{do-while}$$

$$\frac{|A|, |B| < \omega \quad f : A \to B\ \textit{bijective}}{\vDash \{pre\}\ a <\$\ \mathcal{U}(A) \sim b <\$\ \mathcal{U}(B)\ \{(a, b).\ f(a) = b \wedge pre\}}\ \texttt{uniform}$$

$$\frac{b_0, b_1 : \texttt{bool}}{\vDash \{b_0 = b_1\}\ \texttt{assert}\ b_0 \sim \texttt{assert}\ b_1\ \{b_0 = \texttt{true} \wedge b_1 = \texttt{true}\}}\ \texttt{asrt}$$

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

Figure 13. Selected probabilistic relational program logic rules

The rule `seq` relates two sequentially composed commands by relating each of the sub-commands. The `swap` rule states that if a certain relation on memories I is invariant w.r.t. the execution of c_0 and c_1, then the order in which commands are executed is not relevant. We used the rule `swap` in §2.3 to swap two independent samplings, in that case the invariant I consisted in the equality of memories. The rule `eqDistrL` allows for rewriting code with the same denotation θ, in the sense of §5.1. The `if` rule relates two conditional statements depending on their bodies. 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. 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. [37]. We believe that, with some additional work, their ideas could be used to also support unbounded loops in SSProve.

The rule `uniform` relates sampling from uniform distributions on finite sets A and B that are in a bijective correspondence. The `asrt` rule relates two `assert` commands, as long as "$b_0 = b_1$" holds before the commands, and guarantees "$b_0 = \texttt{true} \wedge b_0 = \texttt{true}$" afterwards. Finally `asrtL` is an asynchronous variant of `asrt` that specifies the behavior of `assert`, by relating it with `return ()` when the boolean involved in the assert is `true`.

4.2 Proof sketch for Theorem 1

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

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

When we want to prove that two packages with the same interface are equivalent w.r.t. perfect indistinguishability, 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 invariantness of the predicate is the reason why ψ appears both in the pre- and postcondition from Theorem 1.

Before giving the proof sketch for Theorem 1, we postulate a theorem that is also proved in Coq and relates the probabilistic relational program logic with the probabilistic semantics.

Theorem 2. *If two pieces of code c_0, c_1 are such that*

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

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

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

then we have

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

We are now ready to sketch the proof for Theorem 1.

Proof Sketch of Theorem 1. We want to prove that for each adversary $\mathcal{A}$ we have $\alpha(G_0, G_1)(\mathcal{A}) = 0$, i.e. ,

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

Using the hypothesis and that the predicate ψ is stable, we do induction on the code of the procedure $\mathcal{A}.\mathsf{run}$ to establish

$$\vDash \{\psi\}\ \ (\mathcal{A} \circ G_0)\,.\mathsf{run} \sim \\ (\mathcal{A} \circ G_1)\,.\mathsf{run}\ \{(b_0, b_1).b_0 = b_1 \wedge \psi\}$$

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

$$\vDash \{\psi\}\ \texttt{get l}\ (\lambda\texttt{x.ret x}) \sim \texttt{get l}\ (\lambda\texttt{x.ret x})\ \{(v_0, v_1).v_0 = v_1 \wedge \psi\}$$

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, and we explicitly asked for the adversary memory $\texttt{mem}(\mathcal{A})$ to be disjoint from $\texttt{mem}(G_0)$ and $\texttt{mem}(G_1)$. 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 $b_0 = true \iff b_1 = true$, we know from Theorem 2 that

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

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 follows the effect-modular framework [37], instantiating it with probabilities, simple failures, and global state. We first give in §5.1 an overview of the framework of Maillard et al. [37]. We then explain how we apply it to (1) obtain modularly a model for a probabilistic relational program logic in §5.2 and (2) enrich it with state in §5.3.

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. 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 specification w. The exact shape of code and specifications appearing in such a judgment can vary depending on what logic is considered (see Figure 13 for an example).

The recipe laid out by Maillard et al. [37] stems from the realization that not only effectful code can be modeled using monads, but specifications can too, and we can build semantics for $r\mathcal{L}$ using a so-called *relational effect observation* in 3 steps:

1) Model the effects involved in the considered left and right programs as monads M_0 and M_1.
2) Turn the collection of relational specifications w into a *relational specification monad* $(A_0, A_1) \mapsto W(A_0, A_1)$ (RSM) ordered by entailment of specifications.
3) Finally, find an appropriate *relational effect observation* θ mapping computations in $M_0\, A_0 \times M_1\, A_1$ to specifications in $W(A_0, A_1)$, preserving the monadic features present on both sides.

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

$$\vDash_\theta c_0 \sim c_1 \ \{ w \} \Leftrightarrow \theta(c_0, c_1) \leq w$$

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

RSM and effect observation. A RSM W maps a pair of types (A_0, A_1) to a preorder $W(A_0, A_1)$ equipped with operations return and bind at each $(A_0, A_1), (B_0, B_1)$:

$$\texttt{ret}^{\mathrm{W}} : A_0 \times A_1 \to W(A_0, A_1)$$
$$\texttt{bind}^{\mathrm{W}} : W(A_0, A_1) {\to} (A_0 {\times} A_1 {\to} W(B_0, B_1)) {\to} W(B_0, B_1)$$

Even though RSMs do not fit exactly in the usual presentation of a monad, they must satisfy laws similar to the usual identity and associativity monad laws. Moreover, the bind operation should be monotonic with respect to both of its arguments.

A typical example of an RSM 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 ***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). Every pre-/postcondition pair can systematically be translated into a single backward predicate transformer.

A relational effect observation θ between two monads M_0, M_1 and a RSM W is a mapping

$$\theta_{(A_0,A_1)} : M_0 A_0 \times M_1 A_1 \to W(A_0, A_1)$$

laxly preserving the return and bind operations:

$$\theta\,(\texttt{ret}^{M_0}\, a_0, \texttt{ret}^{M_1}\, a_1) \leq \texttt{ret}^{W}\,(a_0, a_1)$$
$$\theta\,(\texttt{bind}^{M_0}\, m_0\, f_0, \texttt{bind}^{M_1}\, m_1\, f_1) \leq \texttt{bind}^{\mathrm{W}}\, \theta\,(m_0, m_1)\ (\theta \circ (f_0, f_1))$$

The second inequation can be understood as a semantic formulation of the seq rule defined in Figure 13. The validity proof for this rule in our model relies directly on this inequation.

In our Coq formalization RSMs and relational effect observations are defined through abstract algebraic structure (that of order-enriched relative monad and suitable morphism of such) from which the description given here can easily be derived.

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 over a probabilistic signature noted F_{Pr}, reusing $\texttt{code}_{\mathcal{L},I}$ 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 [4, 30], following the track of §3.2, but ignoring considerations around state. The semantics assignment can in fact be seen as a monad morphism $\mathrm{F}_{Pr} \to \mathrm{SD}$.

Specifications and effect observation. To model specifications for probabilistic code we use the backward predicate transformer RSM given by $\mathrm{BP}(A_0, A_1) := (A_0 {\times} A_1 {\to} \mathbb{P}) {\to} \mathbb{P}$. The relational effect observation θ_{Pr} is based on probabilistic couplings, as explained in §4.1. For $c_i{:}\mathrm{F}_{Pr}(A_i)$ and $d_i{:}\mathrm{SD}(A_i)$ the associated subdistributions $(i = 0, 1)$ we set:

$$\theta_{Pr}^{A_0 A_1}(c_0, c_1) = \lambda(\phi : A_0 \times A_1 \to \mathbb{P}).\ \exists d \sim_{d_1}^{d_0} \forall a_0 a_1.\ d(a_0, a_1){>}0 \Rightarrow \phi(a_0, a_1)$$

Our probabilistic model $\vDash_{\theta_{Pr}} c_0 \sim c_1 \ \{ w \}$ validates state-free accounts of several rules of Figure 13. 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 $a <\$\ \mathcal{U}(A)$) are validated as well. Third, failing assertions at type A can be modeled using the zero subdistribution on A, and this interpretation allows us to validate the `assert` rule in our model. Fourth, a state-free variant of the `reflexivity` can be established by building, for any subdistribution s, a coupling $d \sim^{s}_{s}$ of s with itself.

5.3 Adding state

In order to extend this first model to stateful code and state-aware specifications, we adapt to our setting the classical notion of *state monad transformer* [33]. A monad transformer maps monads M to monads $\mathrm{T}\,M$ and monad morphisms θ to monad morphisms $\mathrm{T}\,\theta$. Besides, it comes equipped with a family of liftings $\forall M, M \to \mathrm{T}\,M$. We generalize this, and build modularly an effect observation $\theta_{\mathit{Pr,St}}$ on top of θ_{Pr}:

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

where (S_i is the left or right set of global states):

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

To comply with what was done for θ_{Pr} in §5.2 we further extend $\theta'_{\mathit{Pr,St}}$ by turning its domain into a product of free monads $\mathrm{F}^2_{\mathit{Pr,St}}$ over a stateful and probabilistic signature. $\mathrm{F}_{\mathit{Pr,St}}$ actually refers to the monad evoked in §3.2. The said extension is achieved as a precomposition of $\theta'_{\mathit{Pr,St}}$ with the mapping mentioned in §3.2: $\theta_{\mathit{Pr,St}} := \theta'_{\mathit{Pr,St}} \circ \mathtt{Pr\_code}$.

From an abstract point of view, the existence and correctness of the state monad transformer relies on a delicate piece of abstract category theory: we can obtain $\mathrm{StT}\,m$ for a j-relative monad m by simply pasting an adequate 2-cell $(*)$ induced by a j-relative adjunction that exists whenever I is Cartesian, C is Cartesian closed and j preserves Cartesian products.

$$\begin{tikzcd}
 & I \arrow[r, "j"] \arrow[r, bend left=60, "m"] \arrow[r, phantom, shift left=3, "\mathtt{ret}\Uparrow"] & C \arrow[dr, "jS\Rightarrow -"] & \\
I \arrow[ur, "S\times -"] \arrow[rrr, "j"'] \arrow[rrr, phantom, shift left=4, "(*)\Uparrow"] & & & C
\end{tikzcd}$$

Because this construction is modular, we can prove that the final model recasts exactly the model devised in §5.2 when restricted to code and specifications not using state. In other words there is an operation lifting every semantic judgment from the previous model θ_{Pr} to the current model $\theta_{\mathit{Pr,St}}$. This modularity is moreover reflected in the way $\theta_{\mathit{Pr,St}}(c_0, c_1)$ evaluates. A first pass converts stateful operations of c_0, c_1 and yields state-transforming 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 $\theta_{\mathit{Pr,St}}(c_0, c_1)$: $\mathrm{StT}(\mathrm{BP})(A_0, A_1)$. Finally, $\theta_{\mathit{Pr,St}}$ validates all of the rules of our relational program logic (including Figure 13).

A substantial amount of work was required for the implementation of our model in Coq. To establish the existence of the first layer θ_{Pr} of the model, we developed a mathematical theory of couplings and of their interaction with probabilistic programs. This theory relies internally on the `mathcomp-analysis` library, particularly on their formalization of real numbers, subdistributions and discrete integrals. The custom state transformer we develop for $\theta_{\mathit{Pr,St}}$ is built upon the formalization of several non-standard categorical constructs in an order-enriched context: lax functors, lax natural transformations, left relative adjunctions, lax morphisms between such adjunctions.

6 Related Work

SSProve is the first verification framework for SSP, yet the formal verification of cryptographic proofs in different styles has been intensely investigated [5]. In this section we survey the closest related work in this space.

CertiCrypt [9] is a foundational Coq framework for game-based crypto 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 [43] is a more recent foundational Coq framework for crypto proofs that was used to verify the HMAC implementations in OpenSSL [17] and mbedTLS [55]. 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 probabilities monad when needed (§3.2 and §5.2), all without leaving the internal language of Coq. We are not aware of any formalization of SSP on top of FCF, although it seems possible in principle.

EasyCrypt [10, 12] is a proof assistant and verification tool specifically designed and built from scratch for game-based crypto proofs. This state-of-the-art tool has been used, for instance, to prove security for Amazon Web Services' Key Management Service [2]. 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 [6]. EasyCrypt's parameterized modules are, however, quite different from parameterized games in SSP (parameterized module instantiation in EasyCrypt has cloning semantics, i.e., each instance gets a separate copy of the module's state). Moreover, EasyCrypt functors—which can to some extent be used to represent packages with imports—are not first class, so SSP-style laws cannot even be stated. While none of these is a showstopper, it leads to a quite different default style for writing modular proofs.

In very recent work, Dupressoir et al. [27] show that with enough workarounds they can code up in EasyCrypt the SSP proof of Brzuska et al. [22] for the Cryptobox [18] KEM-DEM [26], and discuss the strengths and shortcomings of EasyCrypt for formalizing SSP-style proofs. Our examples are for now less complex, but instead we focus on providing a *general framework* for SSP proofs, including definitions of SSP packages, their composition, and the corresponding algebraic laws. SSProve also includes an `assert` operation, and a faithful representation of the SSP memory model, allowing to express SSP proofs more naturally. We have recently started formalizing in SSProve Brzuska et al.'s [22] proof of Cryptobox, hoping for a direct comparison.

EasyUC [24] aims to address the lack of composability in game-based proofs by formalizing the Universal Composability (UC) framework [23] using EasyCrypt. EasyUC replaces the interactive Turing machines in UC with EasyCrypt modules. It was used to prove a secure messaging protocol composed of Diffie-Hellman and one-time pad. More recent work develops a DSL [25] 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. [6] add automatic complexity analysis to EasyCrypt and use it for another formalization of UC. SSP was in part inspired by the UC framework, but focuses on making game-based proofs more modular and scalable, without targeting simulation-based security or universal composability. A more precise comparison between SSP and UC proofs would be interesting.

CryptHOL [15] is a foundational framework for game-based proofs that uses the theory of relational parametricity to achieve automation in the Isabelle/HOL proof assistant. It also makes use of the extensive mathematical libraries of Isabelle/HOL. More proof engineering and automation would be needed for SSProve to have a chance at matching CryptHOL's formalization of ElGamal or PRF-based encryption. CryptHOL [35] has been also used to formalize Constructive Cryptography [38], another composable framework that inspired SSP, and the example of a one-time pad. While there is some similarity between their converters and SSP's packages, to our knowledge a more precise comparison has not yet been undertaken.

ILC [34] is a process calculus modeling some of the key ideas behind the UC framework, in particular its co-routine based communication mechanism, while completely abstracting away from interactive Turing machines. Their work has not yet been formalized in a proof assistant.

IPDL [42] is another recent Coq framework for crypto proofs. Although their motivation is similar to SSP and their interaction sets are reminiscent of packages, the relation to other composable frameworks has not been worked out.

Packages have been motivated by ML modules [47], but for stateful programs. No specific theory for stateful modules seems to be available, but Sterling and Harper [51] provide a general module system. It would be interesting to specialize it to stateful programs and compare it to packages.

7 Future Work

The high-level proofs done on paper in the miTLS project [19, 20, 29] were the main inspiration for the SSP methodology and it would be an interesting challenge to scale SSProve to large machine-checked proofs in the future. This would for a start require more work on proof engineering and automation. The problem of verifying such large proofs all the way down to low-level efficient executable code is even more challenging, also given the extreme scale of a complete implementation for a protocol like TLS. Achieving this in Coq would probably require integrating with projects such as VST [3] or FiatCrypto [28].

An alternative would be to port SSProve to F⋆ [52], where at least functional correctness can be verified at that scale. Still many challenges would remain, including extending F⋆ to probabilistic verification, internalizing F⋆ modules, and extending the SSP methodology to support type abstraction and procedures with specifications. In less ambitious recent work that is still unfinished, Kohbrok et al. [32] have implemented vanilla SSP packages in F⋆ and attempted to automate state-separating proofs based on a library for partial setoids.

In the shorter term, it would be good to apply SSProve to more proofs and to improve proof engineering in order to make the user experience more pleasant. We would also like to extend SSProve to extra side-effects such as non-termination and I/O and also to F⋆-style sub-effecting [52]. The effect-modular semantic model from §5 should make this easier, and we hope to be able reuse the Interaction Trees framework [50, 54], and maybe also take inspiration in CryptHOL [15].

Acknowledgments

We are grateful to Arthur Azevedo de Amorim, Théo Laurent, Nikolaj Sidorenco, and Ramkumar Ramachandra for their technical support and for participating in stimulating discussions. 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), and by the Concordium Blockchain Research Center at Aarhus University. Antoine Van Muylder holds a PhD Fellowship from the Research Foundation – Flanders (FWO).

References

[1] R. Affeldt, C. Cohen, M. Kerjean, A. Mahboubi, D. Rouhling, K. Sakaguchi, and P.-Y. Strub. mathcomp-analysis. Analysis library compatible with Mathematical Components, 2021.

[2] J. B. Almeida, M. Barbosa, G. Barthe, M. Campagna, E. Cohen, B. Grégoire, V. Pereira, B. Portela, P. Strub, and S. Tasiran. A machine-checked proof of security for AWS key management service. In L. Cavallaro, J. Kinder, X. Wang, and J. Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. 2019.

[3] A. W. Appel. Verified software toolchain - (invited talk). In G. Barthe, editor, *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*. 2011.

[4] P. Audebaud and C. Paulin-Mohring. Proofs of randomized algorithms in coq. In T. Uustalu, editor, *Mathematics of Program Construction*. 2006.

[5] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao, and B. Parno. Sok: Computer-aided cryptography. *IACR Cryptol. ePrint Arch.*, 2019, 2019.

[6] M. Barbosa, G. Barthe, B. Grégoire, A. Koutsos, and P.-Y. Strub. Mechanized proofs of adversarial complexity and application to universal composability. Cryptology ePrint Archive, Report 2021/156, 2021.

[7] R. Barnes, B. Beurdouche, J. Millican, E. Omara, K. Cohn-Gordon, and R. Robert. The messaging layer security (MLS) protocol. IETF Draft, 2020.

[8] G. Barthe, B. Grégoire, and S. Zanella-Béguelin. Formal certification of code-based cryptographic proofs. *POPL*, 2009.

[9] G. Barthe, B. Grégoire, and S. Zanella Béguelin. Formal certification of code-based cryptographic proofs. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2009.

[10] G. Barthe, B. Grégoire, S. Heraud, and S. Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In P. Rogaway, editor, *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*. 2011.

[11] G. Barthe, J. M. Crespo, B. Grégoire, C. Kunz, Y. Lakhnech, B. Schmidt, and S. Zanella Béguelin. Fully automated analysis of padding-based encryption in the computational model. In A. Sadeghi, V. D. Gligor, and M. Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*. 2013.

[12] G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P. Strub. EasyCrypt: A tutorial. In A. Aldini, J. Lopez, and F. Martinelli, editors, *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*. 2013.

[13] G. Barthe, T. Espitau, B. Grégoire, J. Hsu, L. Stefanesco, and P. Strub. Relational reasoning via probabilistic coupling. In *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, 2015.

[14] G. Barthe, B. Grégoire, and B. Schmidt. Automated proofs of pairing-based cryptography. In I. Ray, N. Li, and C. Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*. 2015.

[15] D. A. Basin, A. Lochbihler, and S. R. Sefidgar. CryptHOL: Game-based proofs in higher-order logic. *J. Cryptol.*, 33(2), 2020.

[16] M. Bellare and P. Rogaway. Code-based game-playing proofs and the security of triple encryption. *IACR Cryptol. ePrint Arch.*, 2004, 2004.

[17] L. Beringer, A. Petcher, K. Q. Ye, and A. W. Appel. Verified correctness and security of OpenSSL HMAC. In J. Jung and T. Holz, editors, *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*. 2015.

[18] D. J. Bernstein. Cryptography in NaCl, 2009.

[19] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, P. Strub, and S. Zanella Béguelin. Proving the TLS handshake secure (as it is). In J. A. Garay and R. Gennaro, editors, *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II*. 2014.

[20] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Pan, J. Protzenko, A. Rastogi, N. Swamy, S. Zanella Béguelin, and J. K. Zinzindohoue. Implementing and proving the TLS 1.3 record layer. *IEEE Security & Privacy*, 2017.

[21] B. Blanchet. A computationally sound mechanized prover for security protocols. In *2006 IEEE Symposium on Security and Privacy (S&P 2006), 21-24 May 2006, Berkeley, California, USA*. 2006.

[22] C. Brzuska, A. Delignat-Lavaud, C. Fournet, K. Kohbrok, and M. Kohlweiss. State separation for code-based game-playing proofs. In T. Peyrin and S. Galbraith, editors, *Advances in Cryptology – ASIACRYPT 2018*. 2018.

[23] R. Canetti. Universally composable security. *J. ACM*, 67 (5), 2020.

[24] R. Canetti, A. Stoughton, and M. Varia. EasyUC: Using EasyCrypt to mechanize proofs of universally composable security. In *CSF*. 2019.

[25] R. Canetti, A. Kfoury, A. Stoughton, M. Varia, G. Tarakaram, and T. Petrovic. UC Domain Specific Language. unpublished, 2021.

[26] R. Cramer and V. Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM J. Comput.*, 33(1), 2003.

[27] F. Dupressoir, K. Kohbrok, and S. Oechsner. Bringing state-separating proofs to EasyCrypt - a security proof for Cryptobox. Cryptology ePrint Archive, Report 2021/326, 2021.

[28] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. *IEEE S&P*, 2019.

[29] C. Fournet, M. Kohlweiss, and P. Strub. Modular code-based cryptographic verification. *CCS*. 2011.

[30] M. Giry. A categorical approach to probability theory. *Categorical Aspects of Topology and Analysis*. 1982.

[31] S. Halevi. A plausible approach to computer-aided

cryptographic proofs. *IACR Cryptol. ePrint Arch.*, 2005, 2005.

[32] K. Kohbrok, M. Kohlweiss, T. Ramananandro, and N. Swamy. Relational F* for state separating cryptographic proofs. F* wiki article, 2020.

[33] S. Liang, P. Hudak, and M. P. Jones. Monad transformers and modular interpreters. *POPL*. 1995.

[34] K. Liao, M. A. Hammer, and A. Miller. ILC: a calculus for composable, computational cryptography. In *PLDI*. 2019.

[35] A. Lochbihler, S. R. Sefidgar, D. A. Basin, and U. Maurer. Formalizing constructive cryptography using CryptHOL. In *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*. 2019.

[36] A. Mahboubi and E. Tassi. Mathematical components. Online book, 2021.

[37] K. Maillard, C. Hritcu, E. Rivas, and A. V. Muylder. The next 700 relational program logics. *Proc. ACM Program. Lang.*, 4(POPL), 2020.

[38] U. Maurer. Constructive cryptography - A new paradigm for security definitions and proofs. In S. Mödersheim and C. Palamidessi, editors, *Theory of Security and Applications - Joint Workshop, TOSCA 2011, Saarbrücken, Germany, March 31 - April 1, 2011, Revised Selected Papers*. 2011.

[39] U. Maurer and R. Renner. Abstract cryptography. In B. Chazelle, editor, *Innovations in Computer Science - ICS 2011, Tsinghua University, Beijing, China, January 7-9, 2011. Proceedings*. 2011.

[40] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1), 1992.

[41] E. Moggi. Computational lambda-calculus and monads. *LICS*. 1989.

[42] G. Morrisett, E. Shi, K. Sojakova, X. Fan, and J. Gancher. IPDL: A simple framework for formally verifying distributed cryptographic protocols. Cryptology ePrint Archive, Report 2021/147, 2021.

[43] A. Petcher and G. Morrisett. The foundational cryptography framework. *POST*. 2015.

[44] G. D. Plotkin and J. Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1), 2003.

[45] G. D. Plotkin and M. Pretnar. Handlers of algebraic effects. *ESOP*. 2009.

[46] E. Rescorla. The transport layer security (TLS) protocol version 1.3. IETF RFC 5246, 2018.

[47] A. Rossberg, C. V. Russo, and D. Dreyer. F-ing modules. *J. Funct. Program.*, 24(5), 2014.

[48] M. Rosulek. The joy of cryptography. Online textbook, 2021.

[49] V. Shoup. Sequences of games: a tool for taming complexity in security proofs. *IACR Cryptol. ePrint Arch.*, 2004, 2004.

[50] L. Silver and S. Zdancewic. Dijkstra monads forever: termination-sensitive specifications for interaction trees. *Proc. ACM Program. Lang.*, 5(POPL), 2021.

[51] J. Sterling and R. Harper. Logical relations as types: Proof-relevant parametricity for program modules. *CoRR*, abs/2010.08599, 2020.

[52] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoué, and S. Zanella-Béguelin. Dependent types and multi-monadic effects in F*. *POPL*. 2016.

[53] P. Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*. 1990.

[54] L. Xia, Y. Zakowski, P. He, C. Hur, G. Malecha, B. C. Pierce, and S. Zdancewic. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.*, 4(POPL), 2020.

[55] K. Q. Ye, M. Green, N. Sanguansin, L. Beringer, A. Petcher, and A. W. Appel. Verified correctness and security of mbedTLS HMAC-DRBG. In B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. 2017.