Programming language techniques for high-assurance cryptography

Suggested Citation: Gilles Barthe (2017), "Programming language techniques for high-assurance cryptography", : Vol. xx, No. xx, pp 1–1. DOI: 10.1561/XXXXXXXXX.

Gilles Barthe

now
the essence of knowledge
Boston — Delft

Contents

Programming language techniques for high-assurance cryptography

Gilles Barthe[1]

[1] *IMDEA Software Institute*

ABSTRACT

Gilles Barthe (2017), "Programming language techniques for high-assurance cryptography", : Vol. xx, No. xx, pp 1–1. DOI: 10.1561/XXXXXXXXX.

1

Introduction to probabilistic couplings

1.1 Distributions

For simplicity, we only consider sub-distributions with a discrete support.

Definition 1.1 (Sub-distributions)**.** A (discrete) sub-distribution over a set A is a function $\mu : A \to [0, 1]$ such that:

- the *support* $\mathrm{supp}(\mu) = \{a \in A \mid \mu(a) > 0\}$ of μ is a discrete set;
- the *mass* $|\mu| = \sum_{a \in A} \mu(a)$ of μ is defined and verifies $|\mu| \leq 1$.

Sub-distributions of mass 1 are called *(full) distributions*; other sub-distributions are called *proper sub-distributions*. The set of sub-distributions over A is denoted by $\mathbb{D}(A)$.

Sub-distributions are partially ordered by the pointwise inequality inherited from reals: given any two sub-distributions $\mu, \mu' \in \mathbb{D}(A)$, we have $\mu \leq \mu'$ iff $\mu(a) \leq \mu'(a)$ for every $a \in A$. Moreover, two sub-distributions are equal iff they assign the same value (i.e., probability) to each element in their domain: given any two sub-distributions $\mu, \mu' \in \mathbb{D}(A)$, we have $\mu = \mu'$ iff $\mu(a) = \mu'(a)$ for every $a \in A$. Note that $\mu = \mu'$ iff $\mu \leq \mu'$ and $\mu' \leq \mu$.

The following lemma proves that equality and inequality coincide for distributions.

Lemma 1.1. Let $\mu, \mu' \in \mathbb{D}(A)$.

- If $\mu \leq \mu'$ then $|\mu| \leq |\mu'|$.
- If $|\mu| = 1$ and $\mu \leq \mu'$ then $\mu = \mu'$.

1.2 Events and probabilities

Events over a set A are subsets of A. The probability of an event E w.r.t. a sub-distribution μ, written as $\mathbb{P}_\mu[E]$, is defined as $\sum_{x \in E} \mu(x)$. We almost exclusively use the following basic facts about events:

- for every event E, $0 \leq \mathbb{P}_\mu[E] \leq 1$;
- $\mathbb{P}_\mu[\bot] = 0$;
- $\mathbb{P}_\mu[\top] = 1$ if μ is a full distribution;
- for every events E and F, $E \subseteq F$ implies $\mathbb{P}_\mu[E] \leq \mathbb{P}_\mu[F]$;
- for every events E and F, $\mathbb{P}_\mu[E \cup F] = \mathbb{P}_\mu[E] + \mathbb{P}_\mu[F] - \mathbb{P}_\mu[E \cap F]$.

1.3 Couplings

Informally, a probabilistic coupling for two sub-distributions $\mu_1 \in \mathbb{D}(A_1)$ and $\mu_2 \in \mathbb{D}(A_2)$ is a sub-distribution $\mu \in \mathbb{D}(A_1 \times A_2)$ which lets the two sub-distributions share some randomness. Formally, the definition of probabilistic coupling requires that the left and right projections of μ (also known as its marginals, and defined formally below), coincide with μ_1 and μ_2 respectively.

Definition 1.2 (Marginal sub-distributions)**.** The first and second marginals of a sub-distribution $\mu \in \mathbb{D}(A_1 \times A_2)$ are the two sub-distributions $\pi_1(\mu) \in \mathbb{D}(A_1)$ and $\pi_2(\mu) \in \mathbb{D}(A_2)$ given by

$$\pi_1(\mu)(a_1) = \sum_{a_2 \in A_2} \mu(a_1, a_2) \qquad \pi_2(\mu)(a_2) = \sum_{a_1 \in A_1} \mu(a_1, a_2)$$

For our purposes, we are interested in probabilistic couplings that lie within a given relation, to be thought as the post-condition of the coupling.

Definition 1.3 (Probabilistic couplings)**.** Let $R \subseteq A_1 \times A_2$ be a binary relation. Let $\mu_1 \in \mathbb{D}(A_1)$ and $\mu_2 \in \mathbb{D}(A_2)$. A R-coupling for μ_1 and μ_2 is a discrete probability sub-distribution $\mu \in \mathbb{D}(A_1 \times A_2)$ such that the following conditions are satisfied:

- **marginals:** $\pi_1(\mu) = \mu_1$ and $\pi_2(\mu) = \mu_2$
- **support:** $\mathrm{supp}(\mu) \subseteq R$

We write $\mu \blacktriangleleft_R \langle \mu_1 \,\&\, \mu_2 \rangle$ whenever μ is a R-coupling of μ_1 and μ_2.

For many purposes, it suffices to know the existence of a R-coupling. This leads to the definition of R-lifting.

Definition 1.4 (Probabilistic lifting of a relation)**.** Let $R \subseteq A_1 \times A_2$ be a binary relation. The probabilistic lifting of the relation R is the binary relation $R^\sharp$ over $\mathbb{D}(A_1) \times \mathbb{D}(A_2)$ such that for every $\mu_1 \in \mathbb{D}(A_1)$ and $\mu_2 \in \mathbb{D}(A_2)$, $(\mu_1, \mu_2) \in R^\sharp$ iff there exists μ such that $\mu \blacktriangleleft_R \langle \mu_1 \,\&\, \mu_2 \rangle$.

1.3.1 Properties of couplings

The following is an important consequence of the definition of R-couplings.

Lemma 1.2. If $\mu \blacktriangleleft_R \langle \mu_1 \,\&\, \mu_2 \rangle$ then $|\mu| = |\mu_1| = |\mu_2|$.

Proof. We do the proof only for $\mu_1 = \mu$

$$\begin{array}{rcll} |\mu_1| & = & \sum_{a_1 \in A_1} \mu_1(a_1) & \text{(by definition)} \\ & = & \sum_{a_1 \in A_1} \pi_1(\mu)(a_1) & \text{(marginals)} \\ & = & \sum_{a_1 \in A_1} \sum_{a_2 \in A_2} \mu(a_1, a_2) & \text{(by definition)} \\ & = & \sum_{(a_1,a_2) \in A_1 \times A_2} \mu(a_1, a_2) & \\ & = & |\mu| & \text{(by definition)} \end{array}$$

□

Conversely, $\top$-couplings always exist for pairs of sub-distributions whose mass coincide.

Lemma 1.3 (Existence of $\top$-couplings)**.** Let $\mu_1 \in \mathbb{D}(A_1)$ and $\mu_2 \in \mathbb{D}(A_2)$. If $|\mu_1| = |\mu_2|$ then there exists $\mu \in \mathbb{D}(A_1 \times A_2)$ such that $\mu \blacktriangleleft_\top \langle \mu_1 \,\&\, \mu_2 \rangle$.

Proof. We need to exhibit a sub-distribution μ over $A_1 \times A_2$ with the desirable properties. It suffices to take a rescaling of the product distribution:

$$\mu(a_1, a_2) = \frac{\mu_1(a_1) \cdot \mu_2(a_2)}{|\mu_1|}$$

□

There exist many $\top$-couplings for probability sub-distributions with the same mass; one interesting instance is the *optimal coupling*, when $A_1 = A_2$ (see Exercise 6).

On the other hand, $\bot$-couplings do not exist, except for the null sub-distributions.

Lemma 1.4 (Non-existence of $\bot$-couplings)**.** Let $\mu_1 \in \mathbb{D}(A_1)$ and $\mu_2 \in \mathbb{D}(A_2)$. If $\mu \blacktriangleleft_\bot \langle \mu_1 \,\&\, \mu_2 \rangle$ then $|\mu_1| = |\mu_2| = 0$.

Proof. The sub-distribution μ over $A_1 \times A_2$ must necessarily be the null sub-distribution. By the marginal conditions both μ_1 and μ_2 must also be the null sub-distributions. □

The next lemma shows that R-couplings are preserved under weakenings.

Lemma 1.5 (Monotonocity of couplings)**.** Let $\mu_1 \in \mathbb{D}(A_1)$ and $\mu_2 \in \mathbb{D}(A_2)$ and $\mu \in \mathbb{D}(A_1 \times A_2)$. Moreover let $R, S \subseteq A_1 \times A_2$ such that If $R \subseteq S$. If $\mu \blacktriangleleft_R \langle \mu_1 \,\&\, \mu_2 \rangle$ then $\mu \blacktriangleleft_S \langle \mu_1 \,\&\, \mu_2 \rangle$.

Proof. The marginal conditions follow immediately from the assumption, the support condition follows by transitivity of $\subseteq$. □

Conversely, one can strengthen the relation of the coupling, under suitable conditions.

Lemma 1.6 (Strengthening couplings)**.** Let $\mu_1 \in \mathbb{D}(A_1)$ and $\mu_2 \in \mathbb{D}(A_2)$ and $\mu \in \mathbb{D}(A_1 \times A_2)$. Moreover let $R, S \subseteq A_1 \times A_2$ such that $R \cap (\mathrm{supp}(\mu_1) \times \mathrm{supp}(\mu_2)) \subseteq S$. If $\mu \blacktriangleleft_R \langle \mu_1 \,\&\, \mu_2 \rangle$ then $\mu \blacktriangleleft_S \langle \mu_1 \,\&\, \mu_2 \rangle$.

Proof. The marginal conditions follow immediately from the assumption. For the support condition, the marginal conditions entail that $\mu(a_1, a_2) = 0$ whenever $a_1 \notin \mathrm{supp}(\mu_1)$ or $a_2 \notin \mathrm{supp}(\mu_2)$, so $\mathrm{supp}(\mu) \subseteq (\mathrm{supp}(\mu_1) \times \mathrm{supp}(\mu_2))$. □

The following lemma shows that couplings are not closed under conjunction of relations.

Proposition 1.1. There exists $R, S \subseteq A_1 \times A_2$ and $\mu_1 \in \mathbb{D}(A_1)$ and $\mu_2 \in \mathbb{D}(A_2)$ such that $(\mu_1, \mu_2) \in R^\sharp$ and $(\mu_1, \mu_2) \in S^\sharp$ but $(\mu_1, \mu_2) \notin (R \cap S)^\sharp$.

Proof. Consider $A_1 = A_2 = \{0, 1\}$ and $\mu_1 = \mu_2 = \mathcal{U}_{\{0,1\}}$ be the uniform distribution over booleans. Moreover, let R be the diagonal relation, i.e. for every $(x_1, x_2) \in \{0, 1\} \times \{0, 1\}$, $((x_1, x_2) \in R$ iff $x_1 = x_2$ and let S be its complement, i.e. for every $(x_1, x_2) \in \{0, 1\} \times \{0, 1\}$, $(x_1, x_2) \in S$ iff $x_1 \neq x_2$. We have $\mu \blacktriangleleft_R \langle \mu_1 \,\&\, \mu_2 \rangle$ and $\mu' \blacktriangleleft_S \langle \mu_1 \,\&\, \mu_2 \rangle$, where μ and μ' are defined by the clauses:

$$\mu(x_1, x_2) = \begin{cases} \frac{1}{2} & \text{if } x_1 = x_2 \\ 0 & \text{otherwise} \end{cases} \qquad \mu'(x_1, x_2) = \begin{cases} \frac{1}{2} & \text{if } x_1 \neq x_2 \\ 0 & \text{otherwise} \end{cases}$$

However, there exists no sub-distribution μ'' such that $\mu'' \blacktriangleleft_{R \cap S} \langle \mu_1 \,\&\, \mu_2 \rangle$, since $R \cap S = \emptyset$, and therefore the above would entail $|\mu''| = 0$, which contradicts $\pi_i(\mu'') = \mu_i$ as $|\mu_i| = 1$. □

The next proposition establishes that couplings are also closed under convex combinations.

Lemma 1.7 (Convex combinations of couplings)**.** Let $R \subseteq A_1 \times A_2$, and let I be a finite set. Let $(\mu_1^i)_{i \in I} \in \mathbb{D}(A_1)$, $(\mu_2^i)_I \in \mathbb{D}(A_2)$ and $(\mu^i)_{i \in I} \in \mathbb{D}(A_1 \times A_2)$ such that $\mu^i \blacktriangleleft_R \langle \mu_1^i \,\&\, \mu_2^i \rangle$ for every $i \in I$. For every $(p^i)_{i \in I} \in [0, 1]$ such that $\sum_{i \in I} p_i \leq 1$, we have

$$\sum_{i \in I} p^i \mu^i \blacktriangleleft_R \langle \sum_{i \in I} p^i \mu_1^i \,\&\, \sum_{i \in I} p^i \mu_2^i \rangle$$

where $\sum_{i \in I} p^i \mu^i$ is the sub-distribution defined by the clause $(\sum_{i \in I} p^i \mu^i)(a) = \sum_{i \in I} p^i \mu^i(a)$.

1.4 Bijective couplings

A common strategy to establish a R-coupling, with $R \subseteq A_1 \times A_2$, between two sub-distributions μ_1 and μ_2, is to exhibit a mapping $h : A_1 \to A_2$ that satisfies the following three conditions:

bijectivity: for every $a_1, a_1' \in \mathrm{supp}(\mu_1)$, if $h(a_1) = h(a_1')$ then $a_1 = a_1'$, and for every $a_2 \in \mathrm{supp}(\mu_2)$, there exists $a_1 \in \mathrm{supp}(\mu_1)$ such that $h(a_1) = a_2$;

graph inclusion: for every $a_1 \in \mathrm{supp}(\mu_1)$, $(a_1, h(a_1)) \in R$;

equal probabilities: for every $a \in A_1$,

$$\mathbb{P}_{x_1 \sim \mu_1}[x_1 = a] = \mathbb{P}_{x_2 \sim \mu_2}[x_2 = h(a)]$$

(In particular, this entails that $h(a) \in \mathrm{supp}(\mu_2)$ for every $a \in \mathrm{supp}(\mu_1)$.)

We write $h \blacktriangleleft_R \langle \mu_1 \,\&\, \mu_2 \rangle$ whenever the above conditions are satisfied.

Proposition 1.2. If $h \blacktriangleleft_R \langle \mu_1 \,\&\, \mu_2 \rangle$ then $\mu \blacktriangleleft_R \langle \mu_1 \,\&\, \mu_2 \rangle$, where

$$\mathbb{P}_{x \sim \mu}[x = (a_1, a_2)] = \begin{cases} \mathbb{P}_{x_1 \sim \mu_1}[x_1 = a_1] & \text{if } h(a_1) = a_2 \\ 0 & \text{otherwise} \end{cases}$$

Proof. By inspection. □

There are examples of couplings that cannot be established using bijective couplings. Consider for instance the uniform distributions over bitstrings of length ℓ and $\ell + \ell'$ respectively. The two distributions are R-coupled for the relation R defined by the clause $(x, y) \in R$ iff $x = [y]_\ell$, but there is no bijective coupling between the two distributions.

1.5 From couplings to probabilistic inequalities

R-couplings enjoy many important properties. In what follows, we summarize the properties that are of immediate importance for our context.

Proposition 1.3 (Fundamental theorem of R-couplings)**.** Let $E_1 \subseteq A_1$ and $E_2 \subseteq A_2$. Let $R \subseteq A_1 \times A_2$, such that for every $(a_1, a_2) \in A_1 \times A_2$, $(a_1, a_2) \in R$ implies $a_1 \in E_1 \implies a_2 \in E_2$. If $(\mu_1, \mu_2) \in R^\sharp$ then

$$\mathbb{P}_{\mu_1}[E_1] \leq \mathbb{P}_{\mu_2}[E_2]$$

Proof. Let μ such that $\mu \blacktriangleleft_R \langle \mu_1 \,\&\, \mu_2 \rangle$. By the support property of couplings, we know that $\mathrm{supp}(\mu) \subseteq R$, and hence for every $(x_1, x_2) \in A$ such that $\mu(x_1, x_2) \neq 0$, we have $x_1 \notin E_1$ or $x_2 \in E_2$. It follows that $\mathbb{P}_{(x_1,x_2)\sim\mu}[x_1 \in E_1] \leq \mathbb{P}_{(x_1,x_2)\sim\mu}[x_2 \in E_2]$. By the marginal property of couplings, it follows that $\mathbb{P}_{\mu_1}[E_1] \leq \mathbb{P}_{\mu_2}[E_2]$. □

This theorem has many useful corollaries, which we list below. Many of the corollaries are named after their applications in cryptographic proofs.

Corollary 1.8 (Bridging step)**.** Let $E_1 \subseteq A_1$ and $E_2 \subseteq A_2$. Let $R \subseteq A_1 \times A_2$, such that for every $(a_1, a_2) \in A_1 \times A_2$, $(a_1, a_2) \in R$ implies $a_1 \in E_1 \Leftrightarrow a_2 \in E_2$. If $(\mu_1, \mu_2) \in R^\sharp$ then

$$\mathbb{P}_{\mu_1}[E_1] = \mathbb{P}_{\mu_2}[E_2]$$

In many cases, the two events E_1 and E_2 of interest are the same.

Corollary 1.9 (Bridging step from equivalence relation)**.** Let $A_1 = A_2 = A$ and $R \subseteq A \times A$ be an equivalence relation. If $(\mu_1, \mu_2) \in R^\sharp$ then

$$\mathbb{P}_{\mu_1}[E] = \mathbb{P}_{\mu_2}[E]$$

for every $E \subseteq A$ such that $a_1 \in E \wedge (a_1, a_2) \in R$ implies $a_2 \in E$ for every $a_1, a_2 \in A$.

The next corollary involves two events E_1 and E_2 whose probability we want to connect, and a third event F, called a failure event.

Corollary 1.10 (Failure event)**.** Let $E_1 \subseteq A_1$ and $E_2, F \subseteq A_2$. Define $R \subseteq A_1 \times A_2$ such that for every $(a_1, a_2) \in A_1 \times A_2$, $(a_1, a_2) \in R$ implies $a_1 \in E_1 \implies a_2 \in E_2 \cup F$. If $(\mu_1, \mu_2) \in R^\sharp$ then

$$\mathbb{P}_{\mu_1}[E_1] - \mathbb{P}_{\mu_2}[E_2] \leq \mathbb{P}_{\mu_2}[F]$$

Proof. By proposition 1.3 we have $\mathbb{P}_{\mu_1}[E_1] \leq \mathbb{P}_{\mu_2}[E_2 \cup F]$ and we have $\mathbb{P}_{\mu_2}[E_2 \cup F] \leq \mathbb{P}_{\mu_2}[E_2] + \mathbb{P}_{\mu_2}[F]$, the conclusion follows trivially. □

The next corollary provides a symmetric version of the previous corollary.

Corollary 1.11 (Delayed failure event)**.** Let $E_1, F_1 \subseteq A_1$ and $E_2, F_2 \subseteq A_2$. Define $R \subseteq A_1 \times A_2$ such that for every $(a_1, a_2) \in A_1 \times A_2$, $(a_1, a_2) \in R$ implies $a_1 \in E_1 \cap \neg F_1 \Leftrightarrow a_2 \in E_2 \cap \neg F_2$. If $(\mu_1, \mu_2) \in R^\sharp$ then

$$\left|\mathbb{P}_{\mu_1}[E_1] - \mathbb{P}_{\mu_2}[E_2]\right| \leq \max(\mathbb{P}_{\mu_1}[F_1], \mathbb{P}_{\mu_2}[F_2])$$

By corollary Theorem 1.8, we have $\mathbb{P}_{\mu_1}[E_1 \cap \neg F_1] = \mathbb{P}_{\mu_2}[E_2 \cap \neg F_2]$, by elementary reasoning:

$$\begin{array}{rcl} \left|\mathbb{P}_{\mu_1}[E_1] - \mathbb{P}_{\mu_2}[E_2]\right| & = & \left| \begin{array}{l} \mathbb{P}_{\mu_1}[E_1 \cap F_1] + \mathbb{P}_{\mu_1}[E_1 \cap \neg F_1] \\ -\mathbb{P}_{\mu_2}[E_2 \cap F_2] - \mathbb{P}_{\mu_2}[E_2 \cap \neg F_2] \end{array} \right| \\ & = & \left|\mathbb{P}_{\mu_1}[E_1 \cap F_1] - \mathbb{P}_{\mu_2}[E_2 \cap F_2]\right| \\ & \leq & \max(\mathbb{P}_{\mu_1}[E_1 \cap F_1], \mathbb{P}_{\mu_2}[E_2 \cap F_2]) \\ & \leq & \max(\mathbb{P}_{\mu_1}[F_1], \mathbb{P}_{\mu_2}[F_2]) \end{array}$$

The following proposition states that lifting of equality on the underlying set coincides with equality of distributions, and is useful to fall back on standard probability reasoning when proving existence of couplings.

Proposition 1.4 (Equality coupling)**.** If $A_1 = A_2 = A$, then $(\mu_1, \mu_2) \in =^\sharp$, then $(\mu_1, \mu_2) \in =^\sharp$ iff $\mu_1 = \mu_2$, i.e. for every $E \subseteq A$,

$$\mathbb{P}_{\mu_1}[E] = \mathbb{P}_{\mu_2}[E]$$

Proof. The direct implication follows from the Fundamental Theorem of R-Couplings. The existence of an identity coupling is shown by taking as witness the sub-distribution μ such that

$$\mu(a, b) = \begin{cases} \mu_1(a) & \text{if } a = b \\ 0 & \text{otherwise} \end{cases}$$

□

In addition, the following lemma provides a method to decompose a proof that two distributions are related by an equality coupling into a proof that the two distributions are related by pointwise equality.

Proposition 1.5 (Pointwise equality coupling)**.** For every $\mu_1, \mu_2 \in \mathbb{D}(A)$, $\mu_1 = \mu_2$ iff $(\mu_1, \mu_2) \in R_a^\sharp$ for every $a \in A$, where $(x_1, x_2) \in R_a$ iff $x_1 = a \Leftrightarrow x_2 = a$. If moreover μ_1 is a full-distribution, then it suffices that $(\mu_1, \mu_2) \in S_a^\sharp$ for every $a \in A$, where $(x_1, x_2) \in S_a$ iff $x_1 = a \Rightarrow x_2 = a$.

Proof. The direct implication is trivial, so we consider the reverse implication. By Theorem 1.8, it follows that for every $a \in A$, we have $\mathbb{P}_{x\sim\mu_1}[x = a] = \mathbb{P}_{x\sim\mu_2}[x = a]$, hence the two sub-distributions are equal. In case μ_1 is a full distribution, $(\mu_1, \mu_2) \in S_a^\sharp$ implies $\mathbb{P}_{x\sim\mu_1}[x = a] \leq \mathbb{P}_{x\sim\mu_2}[x = a]$, from which we can again conclude equality of distributons. □

The above proposition can be generalized to accommodate a failure event.

1.6 Couplings are closed under monadic operations

Marginal sub-distributions are a specific case of image distributions, when the function considered is taken to be a projection. In the general case, every $\mu \in \mathbb{D}(A)$ and $f : A \to B \to [0, 1]$ induces a distribution $\mathbb{E}_\mu[f]$, called the *image distribution of* μ *along* f, and defined by the clause:

$$\mathbb{E}_\mu[f](b) \triangleq \sum_{a\in\mathrm{supp}(\mu)} \mu(a) \cdot f(a)(b)$$

Sub-distributions can be given a monadic structure.

Definition 1.5 (Monadic structure of sub-distributions)**.** The unit of the monad is the Dirac distribution, which assigns to every $x \in A$ the *Dirac distribution* $\mathbb{1}_x$ defined by the clause:

$$\mathbb{1}_x(y) \triangleq \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise} \end{cases}$$

The monadic composition takes as input a sub-distribution $\mu \in \mathbb{D}(A)$ and a mapping $M : A \to \mathbb{D}(B)$ and returns a distribution $\mathbb{E}_\mu[M] \in \mathbb{D}(B)$ defined by the clause:

$$\mathbb{E}_\mu[M](b) \triangleq \sum_{a \in \mathrm{supp}(\mu)} \mu(a) M(a)(b)$$

it is the image distribution of μ along M

More generally, the expectation $\mathbb{E}_\mu[f]$ of a function $f : A \to \mathbb{R}^+$ w.r.t. a sub-distribution $\mu \in \mathbb{D}(A)$ is defined as $\sum_{x \in A} \mu(x) f(x)$ when this sum exists, and $+\infty$ otherwise.

We now prove that couplings are closed under monadic unit and monadic composition. This is key to support compositional reasoning and to prove soundness of pRHL.

The following establishes that Dirac distributions of elements related by a relation R are R-coupled.

Lemma 1.12. Let $R \subseteq A_1 \times A_2$ and let $(a_1, a_2) \in R$. Then $\mathbb{1}_{(a_1,a_2)} \blacktriangleleft_R \langle \mathbb{1}_{a_1} \,\&\, \mathbb{1}_{a_2} \rangle$.

The next theorem establishes that couplings are closed under sequential composition.

Theorem 1.13 (Sequential composition of couplings)**.** Let $R \subseteq A_1 \times A_2$ and $S \subseteq B_1 \times B_2$. Let $\mu_1 \in \mathbb{D}(A_1)$, $\mu_2 \in \mathbb{D}(A_2)$, $M_1 : A_1 \to \mathbb{D}(B_1)$ and $M_2 : A_2 \to \mathbb{D}(B_2)$. Assume that:

- $\mu \blacktriangleleft_R \langle \mu_1 \,\&\, \mu_2 \rangle$;
- $M(a_1, a_2) \blacktriangleleft_S \langle M_1(a_1) \,\&\, M_2(a_2) \rangle$ for every $(a_1, a_2) \in A_1 \times A_2$ such that $(a_1, a_2) \in R$.

Then $\mathbb{E}_\mu[M] \blacktriangleleft_S \langle \mathbb{E}_{\mu_1}[M_1] \,\&\, \mathbb{E}_{\mu_2}[M_2] \rangle$.

1.7 Alternative characterizations of couplings

For completeness, we provide two alternative characterizations of couplings. Although we do not use them directly, they are useful to build intuition about couplings.

The first characterization is due to **strassen1965existence**

Theorem 1.14 (Strassen's theorem). Let $R \subseteq A_1 \times A_2$, $\mu_1 \in \mathbb{D}(A_1)$ and $\mu_2 \in \mathbb{D}(A_2)$ such that $|\mu_1| = |\mu_2| = 1$. Then $(\mu_1, \mu_2) \in R^\sharp$ iff $\mathbb{P}_{\mu_1}[X] \leq \mathbb{P}_{\mu_2}[R(X)]$ for every $X \subseteq A_1$, where $R(X) = \{a_2 \in A_2 \mid \exists a_1 \in A_1.\, a_1 \in X \wedge (a_1, a_2) \in R\}$ is the image of X under R.

By rescaling, the characterization readily extends to sub-distributions μ_1 and μ_2 such that $|\mu_1| = |\mu_2|$.

The second characterization is based on the following inductive relation:

$$\frac{(a_1, a_2) \in R}{\mathbb{1}_{(a_1,a_2)} \blacktriangleleft_R \langle \mathbb{1}_{a_1} \,\&\, \mathbb{1}_{a_2} \rangle}\ [\textsc{Dirac}]$$

$$\frac{\forall i \in I.\, \mu^i \blacktriangleleft_R \langle \mu_1^i \,\&\, \mu_2^i \rangle}{\sum_{i \in I} p^i \mu^i \blacktriangleleft_R \langle \sum_{i \in I} p^i \mu_1^i \,\&\, \sum_{i \in I} p^i \mu_2^i \rangle}\ [\textsc{Convex}]$$

Proposition 1.6 (Inductive characterization of R-couplings). $\mu \blacktriangleleft_R \langle \mu_1 \,\&\, \mu_2 \rangle$ iff the validity of the coupling can be established using the inductive relation above.

Note that the soundness of the proof system uses the fact that couplings are closed under convex combinations.

1.8 Further reading

We refer the interested reader to (**coupling-survey**) for a gentle introduction to probabilistic couplings.

1.9 Exercises

1. Variant of failure event lemma. Let S be a relation such that $(a_1, a_2) \in S$ iff $a_1 \in F_1 \Leftrightarrow a_2 \in F_2$ and $a_1 \notin F_1 \implies a_1 \in E_1 \Leftrightarrow a_2 \in E_2$. Prove that

$$\left|\mathbb{P}_{\mu_1}[E_1] - \mathbb{P}_{\mu_2}[E_2]\right| \leq \mathbb{P}_{\mu_1}[F_1]$$

and

$$\left|\mathbb{P}_{\mu_1}[E_1] - \mathbb{P}_{\mu_2}[E_2]\right| \leq \mathbb{P}_{\mu_1}[F_2]$$

2. Lifting of equivalence relations. Let R be an equivalence relation over A. Show that R-lifting coincides with equivalence of quotient distributions are equal, i.e.

$$(\mu_1, \mu_2) \in R^\sharp \Leftrightarrow \mu_1/R = \mu_2/R$$

where μ_1/R and μ_2/R are the images of μ_1 and μ_2 under the canonical mapping $[.] : A \to A/R$. This equivalence is often used in probabilistic process algebra for defining probabilistic bisimulations. We refer the reader to the recent book of Deng (2015) for a detailed account.

3. Composition of liftings. Show the following result, by exhibiting a witness distribution. Then prove the same result using Strassen's theorem.

   Let $R \subseteq A_1 \times A_2$ and $S \subseteq A_2 \times A_3$. Let $\mu_1 \in \mathbb{D}(A_1)$, $\mu_2 \in \mathbb{D}(A_2)$, and $\mu_3 \in \mathbb{D}(A_3)$. Assume that $\mu \blacktriangleleft_R \langle \mu_1 \,\&\, \mu_2 \rangle$ and $\mu' \blacktriangleleft_S \langle \mu_2 \,\&\, \mu_3 \rangle$. Then there exists $\mu'' \in \mathbb{D}(A_1 \times A_3)$ such that $\mu'' \blacktriangleleft_{R \circ S} \langle \mu_1 \,\&\, \mu_3 \rangle$. Hint: take

$$\mu''(a_1, a_3) = \sum_{a_2} \frac{\mu(a_1, a_2) \cdot \mu'(a_2, a_3)}{\mu_2(a_2)}$$

   where by convention $\frac{0}{0} = 0$.

4. Limit coupling. Let $(\mu_n)_{n \in \mathbb{N}} \in \mathbb{D}(A)$ be an increasing sequence of sub-distributions over A, i.e. $\mu_n(a) \leq \mu_{n+1}(a)$ for every $a \in A$ and $n \in \mathbb{N}$. Let $\mu_\infty \in \mathbb{D}(A)$ such that $\mu_\infty(a) = \lim_{n \to \infty} \mu_n(a)$ (note that μ_∞ is well-defined since every bounded increasing sequence of real numbers has a limit; moreover, it satisfies the conditions of sub-distributions.

   Let $R \subseteq A_1 \times A_2$ and $(\mu_n)_{n \in \mathbb{N}}$, $(\nu_n)_{n \in \mathbb{N}}$ two increasing sequences of sub-distributions resp. on A_1 and A_2. If for any $n \in \mathbb{N}$, $(\mu_n, \nu_n) \in R^\sharp$ then $(\mu_\infty, \nu_\infty) \in R^\sharp$.

   (Note: the limit of couplings of two increasing sequences of sub-distributions needs not exist. To see this, find two increasing sequences of sub-distributions $(\mu_1^n)_{n \in \mathbb{N}}$ and $(\mu_2^n)_{n \in \mathbb{N}}$, a sequence

of sub-distributions $(\mu_n)^{n\in\mathbb{N}}$ and two disjoint relations R and S, such that for every even n and odd m,

$$\mu^n \blacktriangleleft_R \langle \mu_1^n \,\&\, \mu_2^n \rangle \qquad \mu^n \blacktriangleleft_S \langle \mu_1^n \,\&\, \mu_2^n \rangle$$

By monotonicity of couplings, $\mu^p \blacktriangleleft_{R\cup S} \langle \mu_1^p \,\&\, \mu_2^p \rangle$ for every $p \in \mathbb{N}$. However, $(\mu_n)_{n\in\mathbb{N}}$ has no limit.)

5. Convex combinations of couplings. Prove Lemma 1.7.

6. Optimal coupling. Let $\mu_1, \mu_2 \in \mathbb{D}(A)$ such that $|\mu_1| = |\mu_2| = 1$. Define the sub-distribution $\mu_0 \in \mathbb{D}(A)$ and $\mu \in \mathbb{D}(A \times A)$ by the clauses:

$$\mu_0(x) = \min(\mu_1(x), \mu_2(x))$$

and

$$\mu(x_1, x_2) = \begin{cases} \mu_0(x_1) & \text{if } x_1 = x_2 \\ \frac{(\mu_1(x_1) - \mu_0(x_1))\ (\mu_2(x_2) - \mu_0(x_2))}{\lambda} & \text{if } x_1 \neq x_2 \end{cases}$$

where λ is the total variation distance between μ_1 and μ_2, i.e.

$$\lambda = \mathsf{TV}(\mu_1, \mu_2) = \max_{E\subseteq A} \left|\mathbb{P}_{\mu_1}[E] - \mathbb{P}_{\mu_2}[E]\right|$$

Prove that μ is a $\top$-coupling for (μ_1, μ_2) and $\mathbb{P}_\mu[E] = \lambda$, where E is defined by the clause $(x_1, x_2) \in E$ iff $x_1 \neq x_2$.

Extend the definition of optimal coupling to sub-distributions with equal mass.

2

Introductory examples

The game-based approach (Shoup, 2004; Halevi, 2005; Bellare and Rogaway, 2006) is a methodology to decompose reductionist proofs into a series of smaller steps. In a nutshell, a game-based proof is structured as a sequence of hops; each hop involves two probabilistic experiments, say G_1 and G_2 and two or more events over the output sub-distribution of G_1 or over the output sub-distribution of G_2, and establishes a probabilistic (in)equality of the following form:

bridging step: $\mathbb{P}_{\mathsf{G}_1}[E_1] = \mathbb{P}_{\mathsf{G}_2}[E_2]$. This imequality is often established for a single event E, i.e. $E = E_1 = E_2$ and arises when reasoning about two games G_1 and G_2 that are perfectly equivalent;

lossy step: $\mathbb{P}_{\mathsf{G}_1}[E_1] \leq \mathbb{P}_{\mathsf{G}_2}[E_2]$. This inequality arises in reduction steps;

failure event step: $\mathbb{P}_{\mathsf{G}_1}[E_1] - \mathbb{P}_{\mathsf{G}_2}[E_2] \leq \mathbb{P}_{\mathsf{G}_2}[F]$. This inequalityis often established for a single event E, i.e. $E = E_1 = E_2$ and arises when reasoning about two games G_1 and G_2 that are conditionally equivalent on a so-called failure event F in the second program;

delayed failure event step: $\left|\mathbb{P}_{\mathsf{G}_1}[E] - \mathbb{P}_{\mathsf{G}_2}[E']\right| \leq \max(\mathbb{P}_{\mathsf{G}_1}[F_1], \mathbb{P}_{\mathsf{G}_2}[F_2])$. This inequalityis often established for a single event E, i.e. $E = E_1 = E_2$ and arises when reasoning about two games G_1 and G_2 that are conditionally equivalent on two failure events F_1 and F_2 in the first and second programs respectively.

By combining these inequalities with non-relational steps, which simplify or compute a closed form for probabilities of the form $\mathbb{P}_{\mathsf{G}}[F]$, one can establish reduction statements as discussed in the previous chapters.

2.1 One-time pad

One-Time Pad is a symmetric encryption scheme, parametrized by a natural number ℓ that determines the size of its key, plaintext and ciphertext spaces, and given by the following triple of algorithms:

Key Generation The key generation algorithm $\mathcal{KG}$ outputs a uniformly distributed key k in $\{0,1\}^\ell$;

Encryption Given a key k and a message $m \in \{0,1\}^\ell$, $\mathcal{E}(k,m)$ outputs the ciphertext $c = k \oplus m$ ($\oplus$ denotes bitwise exclusive-or on bitstrings);

Decryption Given a key k and a ciphertext $c \in \{0,1\}^\ell$, the decryption algorithm outputs the message $m = k \oplus c$.

Shannon (1949) defines perfect secrecy of an encryption scheme by the condition that learning a ciphertext does not change any *a priori* knowledge about the likelihood of messages. For our purposes, it is more convenient to consider an alternative definition of perfect secrecy. We say that $(\mathcal{KG}, \mathcal{E}, \mathcal{D})$ achieves perfect secrecy iff for every messages m_1 and m_2, the distributions of ciphertexts obtained by encrypting respectively m_1 and m_2 with a key k sampled uniformly at random are statistically equal. This can be formalized using the game:

Game $\mathsf{OTP}(m)$:
$k \leftarrow \mathcal{KG}()$;
$c \leftarrow \mathcal{E}(k,m)$;

Perfect secrecy is then defined as follows: for every messages m_1 and m_2, and ciphertexts c_0,

$$\mathbb{P}_{\mathsf{OTP}(m_1)}[c = c_0] = \mathbb{P}_{\mathsf{OTP}(m_2)}[c = c_0]$$

Note that perfect secrecy entails that the advantage of a chosen-plaintext adversary against One-Time Pad (defined using a variant of the IND-CPA game for symmetric encryption) is null.

We now sketch a proof of perfect secrecy for One-Time Pad. The key idea is to "coordinate" the two executions of the game OTP (respectively with parameters m_1 and m_2), so that equality of the two output distributions becomes self-evident. The crucial step here is to "coordinate" the probabilistic assignments between the two executions, in a way which forces the equality of the output distributions. In this case, both executions have one probabilistic assignment $k \xleftarrow{\$} \{0,1\}^\ell$, which we must "coordinate". We follow the idea of probabilistic couplings and require that the two probabilistic assignments "share" their randomness. Although there are many ways to share the randomness, it is generally sufficient that one of the programs samples a value v uniformly, and the other program uses as "random value" a value that is computed deterministically from v. In this case, we apply so-called *optimistic sampling*:

- in the first program, we *sample* the key k uniformly at random;
- in the second program, we *define* the key used for encryption as $k \oplus m_1 \oplus m_2$.

Therefore, the ciphertext returned by the first program is $k \oplus m_1$, whereas the ciphertext returned by the second program is $(k \oplus m_1 \oplus m_2) \oplus m_2$. Since the two expressions are semantically equal, one concludes that the two distributions are equal.

In order to make our argument rigorous, we must however give a precise meaning to "share" randomness, and moreover we must prove that randomness is shared correctly. We make the meaning of sharing randomness precise using *product programs*. Informally, a product program is a program that emulates the behavior of the two executions of

the OTP game:

$$\begin{aligned} k &\leftarrow \mathcal{KG}(); \\ k &\leftarrow k \oplus m_1 \oplus m_2; \\ c &\leftarrow \mathcal{E}(k, m_1); \\ c &\leftarrow \mathcal{E}(k, m_2); \end{aligned}$$

Here blue variables represent variables used in the first execution of the OTP game whereas red variables represent variables used in the second execution of the OTP game.

Product programs have an associated notion of correctness: a product program is correct if the joint output sub-distributions of variables tagged with $\cdot_1$ and $\cdot_2$ respectively are equal with the joint output sub-distributions of variables of $\mathsf{OTP}(m_1)$ and $\mathsf{OTP}(m_2)$ respectively. It is easy to prove that the product program above is correct.

A crucial property of couplings, and a consequence of the fundamental theorem of couplings, which will be introduced shortly, is that for every two games G_1 and G_2, and for every variables x of G_1 and x_2 of G_2 with compatible types, and for every value v of compatible type, we have:

$$\mathbb{P}_{\mathsf{G}_1}[x_1 = v] = \mathbb{P}_{\mathsf{G}_2}[x_2 = v]$$

provided $x_1 = x_2$ holds with probability 1 in the product program. In our example, it is easy to see that the output distribution of the product program satisfies $c = c$ with probability 1, from which we deduce:

$$\mathbb{P}_{\mathsf{OTP}(m_1)}[c = c_0] = \mathbb{P}_{\mathsf{OTP}(m_2)}[c = c_0]$$

as required.

2.2 Hashed ElGamal

Hashed ElGamal is a public-key encryption scheme that can be built from a cyclic group G and family $(H_k)_{k \in K} : G \to \{0,1\}^\ell$ of keyed hash functions mapping elements in G to bitstrings of length ℓ. Let q be the order of G and let g be a generator of G. Key generation, encryption

Adversary $\mathcal{B}(\alpha, \beta, \gamma)$:
$\boldsymbol{k} \xleftarrow{\$} K$;
$(m_0, m_1) \leftarrow \mathcal{A}_1(\alpha)$;
$b \xleftarrow{\$} \{0,1\}$;
$h \leftarrow H_{\boldsymbol{k}}(\gamma)$;
$b' \leftarrow \mathcal{A}_2(\beta, h \oplus m_b)$;
return $b = b'$

Adversary $\mathcal{D}(h)$:
$x \xleftarrow{\$} \mathbb{Z}_q; y \xleftarrow{\$} \mathbb{Z}_q$;
$(m_0, m_1) \leftarrow \mathcal{A}_1(g^x)$;
$b \xleftarrow{\$} \{0,1\}$;
$b' \leftarrow \mathcal{A}_2(g^y, h \oplus m_b)$;
return $b = b'$

Figure 2.1: Constructed adversaries for Hashed ElGamal

and decryption algorithms are defined as follows:

$$\begin{array}{lcl} \mathcal{KG}(\,) & \triangleq & k \xleftarrow{\$} K;\ x \xleftarrow{\$} \mathbb{Z}_q;\ \text{return } ((k,x),(k,g^x)) \\ \mathcal{E}((k,\alpha),m) & \triangleq & y \xleftarrow{\$} \mathbb{Z}_q;\ h \leftarrow H_k(\alpha^y);\ \text{return } (g^y, h \oplus m) \\ \mathcal{D}((k,x),(\beta,\zeta)) & \triangleq & h \leftarrow H_k(\beta^x);\ \text{return } h \oplus \zeta \end{array}$$

The space of public keys is $K \times G$ wheread the space of private keys is $K \times \mathbb{Z}_q$. Moreover, the plaintext space is $\{0,1\}^\ell$ whereas the ciphertext space is $G \times \{0,1\}^\ell$.

Hashed ElGamal achieves chosen-plaintext security (IND-CPA) assuming that the Decisional Diffie-Hellman assumption holds for G and that $(H_k)_{k \in K}$ is entropy-smoothing.

Theorem 2.1 (Chosen-plaintext security of Hashed ElGamal)**.** For every adversary $(\mathcal{A}_1, \mathcal{A}_2)$ against the chosen-plaintest security of Hashed ElGamal, there exists an adversary $\mathcal{D}$ against the entropy-smoothness of H_k and an adversary $\mathcal{B}$ against the decisional Diffie-Hellman for G such that:

$$\left| \mathbb{P}_{\mathsf{IND\text{-}CPA}}[b = b'] - \frac{1}{2} \right| \leq \mathbf{Adv}^{\mathcal{B}}_{\mathsf{DDH}} + \mathbf{Adv}^{\mathcal{D}}_{\mathsf{ES}}$$

The code of the adversaries $\mathcal{D}$ and $\mathcal{B}$ is given in Figure 2.1.

Note that the complexity of the constructed adversaries $\mathcal{D}$ and $\mathcal{B}$ is similar to the complexity of $\mathcal{A}_1$ and $\mathcal{A}_2$.

We review the proof, using the game-based technique. The first step of the proof is to observe that the instantiation of the DDH_0 game

with adversary $\mathcal{B}$ is semantically equivalent to the instantiation of the IND-CPA game for Hashed ElGamal. To support this claim, we build a product program for the DDH_0 and IND-CPA games:

$$
\begin{array}{l}
x \xleftarrow{\$} \mathbb{Z}_q; y \xleftarrow{\$} \mathbb{Z}_q; x \leftarrow x; y \leftarrow y; \\
\boldsymbol{k} \xleftarrow{\$} K; \boldsymbol{k} \leftarrow \boldsymbol{k}; \\
((m_0, m_1), (m_0, m_1)) \leftarrow \mathcal{A}_1(g^x, g^x); \\
b \xleftarrow{\$} \{0,1\}; b \leftarrow b; \\
h \leftarrow H_{\boldsymbol{k}}(g^{xy}); h \leftarrow H_{\boldsymbol{k}}(g^{xy}); \\
v \leftarrow h \oplus m_b; v \leftarrow h \oplus m_b; \\
(b', b') \leftarrow \mathcal{A}_2((g^y, v), (g^y, v)); \\
\mathsf{return}\ (b = b', b = b')
\end{array}
$$

In this case, the two programs make exactly the same number of random samplings, and the product program literally shares randomness; for each random assignment to a variable x in the original program, we perform the same random assignment to the variable x in the left program, and copy the sampled value to x in place of performing a random assignment on the right program.

Moreover, the two programs involve adversary calls, which must thus be handled in the product construction. The solution is simply to let adversaries take two copies of each input, and return two copies of each output. We do not specify the code of adversaries, but require that they return equal outputs if given equal inputs. Concretely, $\mathcal{A}_1$ should be such that $(m_0, m_1) = (m_0, m_1)$ if $x = y$, and $\mathcal{A}_2$ should be such that $b' = b'$ if $(g^y, v) = (g^y, v)$.

The conditions on adversaries entail that $b = b' = b = b'$ holds with probability 1. By the fundamental theorem of couplings, it follows that $\mathbb{P}_{\mathsf{IND\text{-}CPA}}[b = b'] = \mathbb{P}_{\mathsf{DDH}_0}[d = 1]$.

Next, observe that the instantiation of the DDH_1 game with adversary $\mathcal{B}$ is semantically equivalent to the instantiation of the ES_1 game with adversary $\mathcal{D}$. To support this claim, we build a product program similar to the previous step. This entails that $\mathbb{P}_{\mathsf{DDH}_1}[d = 1] = \mathbb{P}_{\mathsf{ES}_1}[d = 1]$.

To prepare for the final steps, observe that inlining the adversary $\mathcal{D}$ in the ES_0 game yield (up to renaming and swapping independent

instructions) the following game:

Game G :
$\boldsymbol{k} \xleftarrow{\$} K; x \xleftarrow{\$} \mathbb{Z}_q; y \xleftarrow{\$} \mathbb{Z}_q;$
$(m_0, m_1) \leftarrow \mathcal{A}_1(g^x);$
$b \xleftarrow{\$} \{0,1\};$
$h \xleftarrow{\$} \{0,1\}^\ell;$
$v \leftarrow h \oplus m_b;$
$b' \leftarrow \mathcal{A}_2(g^y, v);$
return $b = b'$

Therefore, we have $\mathbb{P}_{\mathsf{ES}_1}[d = 1] = \mathbb{P}_{\mathsf{G}}[b = b']$.

Next, we use optimistic sampling to replace the assignments of h and v by semantically equivalent assignments, where this time v is sampled uniformly, and h is defined from v:

Game G' :
$\boldsymbol{k} \xleftarrow{\$} K; x \xleftarrow{\$} \mathbb{Z}_q; y \xleftarrow{\$} \mathbb{Z}_q;$
$(m_0, m_1) \leftarrow \mathcal{A}_1(g^x);$
$b \xleftarrow{\$} \{0,1\};$
$v \xleftarrow{\$} \{0,1\}^\ell;$
$h \leftarrow v \oplus m_b;$
$b' \leftarrow \mathcal{A}_2(g^y, v);$
return $b = b'$

This step of optimistic sampling can be justified in the same way as for one-time pad. We have:

$$\mathbb{P}_{\mathsf{G}}[b = b'] = \mathbb{P}_{\mathsf{G}'}[b = b']$$

At this point it is clear that the challenge does not depend on the bit b and therefore its chances of returning the right guess is one half, i.e. $\mathbb{P}_{\mathsf{G}'}[b = b'] = \frac{1}{2}$. It follows that

$$\mathbb{P}_{\mathsf{ES}_0}[d = 1] = \frac{1}{2}$$

By putting everything together, we get that:

$$\begin{array}{rcl}
\mathbf{Adv}^{\mathcal{A}}_{\mathsf{IND\text{-}CPA}} & = & |\mathbb{P}_{\mathsf{IND\text{-}CPA}}[b = b'] - \frac{1}{2}| \\
& = & |\mathbb{P}_{\mathsf{DDH}_0}[d = 1] - \frac{1}{2}| \\
& \leq & |\mathbb{P}_{\mathsf{DDH}_0}[d = 1] - \mathbb{P}_{\mathsf{DDH}_1}[d = 1]| + |\mathbb{P}_{\mathsf{DDH}_1}[d = 1] - \frac{1}{2}| \\
& \leq & \mathbf{Adv}^{\mathcal{B}}_{\mathsf{DDH}} + |\mathbb{P}_{\mathsf{ES}_1}[d = 1] - \mathbb{P}_{\mathsf{ES}_0}[d = 1]| \\
& \leq & \mathbf{Adv}^{\mathcal{B}}_{\mathsf{DDH}} + \mathbf{Adv}^{\mathcal{D}}_{\mathsf{ES}}
\end{array}$$

This concludes our proof. This example, while simple, exercises two fundamental mechanisms of game-based security proofs: reductions, in which games are instantiated with explicit adversaries and proved equivalent to other games, and semantics-preserving transformations, which restructure or simplify games without modifying their meaning.

2.3 UP TO BAD EXAMPLE

2.4 Discussion

We have sketched how product programs provide a rigorous method for justifying game-based cryptographic proofs. While building product programs explicitly is tractable for small examples, this approach becomes untractable when considering (even slightly) more complex examples. Fortunately, all of our proofs only use the existence of product programs, or equivalently the existence of probabilistic couplings between the two programs. Therefore, our verification methods will leave product programs and probabilistic couplings implicit, and simply assert their existence.

Note that the construction of probabilistic couplings is rarely made explicit in cryptographic papers. To our knowledge, only a handful of cryptographic papers use probabilistic couplings, and never with the aim to justify game-based proofs.

3

Cryptographic games

We now introduce a probabilistic programming language pWhile for writing games. The language is directly inspired from prior work on code-based game-based security proofs and is sufficiently expressive for modeling all the security notions and assumptions introduced in **??** and most cryptographic constructions. For now, our language does not feature adversaries, and is thus not very useful for reasoning about cryptography. In Chapter 6, we present an extension of the language with adversaries.

3.1 Types

Our programming language adopts a strongly-typed discipline. This entails that all expressions must be well-typed, variables must be assigned expressions whose type is compatible with their own type, arguments of of procedure calls must respect the signature of the procedure, and so forth. Although we leave it explicit, types are potentially related by a subtyping relation; for example, the type $\{0,1\}^\ell$ of bitstrings of length ℓ is a subtype of the type $\{0,1\}^*$ of bitstrings of arbitrary length.

Definition 3.1 (Types)**.** Let $\mathcal{T}_{\text{base}}$ be a set of base types and let $\mathcal{CT}$ be

a set of type constructors, such that each element $T \in \mathcal{CT}$ has an arity $k \in \mathbb{N}$. The set $\mathcal{T}$ of types is defined by the following syntax:

$$\begin{array}{rcll} \sigma & ::= & b & \text{base type} \\ & | & T(\sigma_1, \ldots, \sigma_n) & \text{type constructor} \\ & | & \sigma_1 \times \ldots \times \sigma_n & \text{product type} \end{array}$$

Typical examples of type constructors are lists, finite maps, and the error monad, which takes as input a type A and returns a type $A_\perp$ that contains all elements of A, and a distinguished element $\perp$.

Remark 3.1. It is possible to extend the definition of types with a constructor for function types, and to consider a richer type discipline with (predicative) polymorphism. For the sake of clarity, we leave such extensions out of our presentation.

We now define a set-theoretical interpretation of types. Note that in many cases, base types have an intended interpretation; for instance, the type $\{0,1\}$ will be interpreted as the type of booleans, and the type $\mathsf{list}(\{0,1\})$ will be interpreted as the set of boolean-valued lists. In other cases, the interpretation is parametric; for instance, the type $\mathsf{list}(\{0,1\}^\ell)$ will be interpreted as the type of lists of bitstrings of length ℓ for some ℓ, and the type $\mathbb{Z}_q$ will be interpreted as the set of integers modulo q for some q.

Definition 3.2 (Interpretation of types)**.** Suppose given a set-theoretical interpretation $[\![b]\!] \in \mathbf{Set}$ for every base type $b \in \mathcal{T}_{\text{base}}$, and a set-theoretical interpretation $[\![T]\!] : \mathbf{Set}^k \to \mathbf{Set}$ for every type constructor $T \in \mathcal{CT}$ of arity k. The interpretation for types is defined inductively by the clauses:

$$\begin{array}{rcl} [\![\sigma_1 \times \ldots \times \sigma_n]\!] & = & [\![\sigma_1]\!] \times \ldots \times [\![\sigma_n]\!] \\ [\![T(\sigma_1, \ldots, \sigma_n)]\!] & = & [\![T]\!]([\![\sigma_1]\!], \ldots, [\![\sigma_n]\!]) \end{array}$$

3.2 Expressions

Expressions of the language are deterministic and built from variables, constants, and operators. Operators include the usual constants and operations for arithmetic, lists, finite maps, groups, etc.

Definition 3.3 (Expressions). Let $\mathbf{Op} = \mathbf{Op}_0 \sqcup \mathbf{Op}_1$ be a set of operators, where elements of $\mathbf{Op}_0$ are constants and elements of $\mathbf{Op}_1$ are functions. Moreover, let $\mathbf{Vars} = \mathbf{Vars}_{\text{glob}} \sqcup \mathbf{Vars}_{\text{loc}}$ be a set of variables, where elements of $\mathbf{Vars}_{\text{glob}}$ are global variables and elements of $\mathbf{Vars}_{\text{loc}}$ are local variables. The set $\mathbf{Expr}$ of expressions is defined by the following syntax:

$$\begin{array}{rlll} e ::= & x & \text{variable} & \\ \mid & c & \text{constant} & \\ \mid & f(e) & \text{application} & \\ \mid & (e_1, \ldots, e_n) & \text{tuple} & \\ \mid & \pi_i(e) & & \text{projection} \end{array}$$

The set $\text{vars}(e)$ of variables of an expression e is defined inductively by the clauses:

$$\begin{array}{rcl} \text{vars}(x) & = & \{x\} \\ \text{vars}(c) & = & \emptyset \\ \text{vars}(f(e)) & = & \text{vars}(e) \\ \text{vars}((e_1, \ldots, e_n)) & = & \bigcup_{1 \leq i \leq n} \text{vars}(e_i) \\ \text{vars}(\pi_i(e)) & = & \text{vars}(e) \end{array}$$

The substitution $e[e'/x]$ of an expression e' for a variable x in an expression e is defined inductively by the clause:

$$\begin{array}{rcll} x[e'/x] & = & e' & \\ y[e'/x] & = & y & \text{if } y \neq x \\ c[e'/x] & = & c & \\ f(e)[e'/x] & = & f(e[e'/x]) & \\ (e_1, \ldots, e_n)[e'/x] & = & (e_1[e'/x], \ldots, e_n[e'/x]) & \\ \pi_i(e)[e'/x] & = & \pi_i(e[e'/x]) & \end{array}$$

The typing discipline for expressions is parametrized by a set Γ of variable, constant and function declarations. Variable declarations are of the form $x : \sigma$, where $x \in \mathbf{Vars}$ and $\sigma \in \mathcal{T}$. Constant declarations are of the form $c : \sigma$, where $c \in \mathbf{Op}_0$ and $\sigma \in \mathcal{T}$. Function declarations of the form $f : \sigma \to \tau$, where $f \in \mathbf{Op}_1$ and $\sigma, \tau \in \mathcal{T}$. We require that there is exactly one declaration per variable.

We equip expressions with a type system which primarily ensures that functions receive arguments of compatible types. The typing rules

for expressions are straightforward:

$$\frac{(x : \sigma) \in \Gamma}{\vdash x : \sigma}\ [\textsc{Var}]$$

$$\frac{(c : \sigma) \in \Gamma}{\vdash c : \sigma}\ [\textsc{Cst}]$$

$$\frac{\vdash e : \sigma \qquad (f : \sigma \to \tau) \in \Gamma}{\vdash f(e) : \tau}\ [\textsc{Fun}]$$

$$\frac{\vdash e_1 : \sigma_1 \quad \ldots \quad \vdash e_n : \sigma_n}{\vdash (e_1, \ldots, e_n) : \sigma_1 \times \ldots \times \sigma_n}\ [\textsc{Prod}]$$

We next turn to defining the semantics of expressions. The semantics is parametrized by a memory m, which maps variables to elements of the interpretation of their type.

Definition 3.4 (Memory)**.** A memory is a mapping m from the set **Vars** of variables to values, such that for every variable x of type σ, we have $m(x) \in [\![\sigma]\!]$. Each memory m implicitly induces a pair $(m_{\mathrm{glob}}, m_{\mathrm{loc}})$ consisting of a global memory and a local memory respectively; the domain of m_{glob} is the set $\mathbf{Vars}_{\mathrm{glob}}$ of global variables, and the domain of m_{loc} is the set $\mathbf{Vars}_{\mathrm{loc}}$ of local variables. For every procedure $\mathcal{F}$, we assume given a default local memory $m_{\mathrm{def}(\mathcal{F})}$.

We adopt standard notation for memory update: given a memory m, a variable x of type σ and a value $v \in [\![\sigma]\!]$, we let $m[x \leftarrow v]$ denote the unique memory such that for every variable y

$$m[x \leftarrow v](y) = \begin{cases} v & \text{if } x = y \\ m(y) & \text{otherwise} \end{cases}$$

We let **Mem** denote the set of memories.

Note that the set of memories is discrete, whenever the set of variables is fixed, and all types are discrete. However, this is not strictly required to define a semantics for probabilistic programs, since our notion of discrete sub-distribution does not require that the underlying set is discrete.

We now turn to the semantics of expressions.

Definition 3.5 (Semantics of expressions)**.** Suppose given a set-theoretical interpretation $[\![c]\!] \in \bigcap_{\sigma \text{s.t.} c:\sigma} [\![\sigma]\!]$ for each constant $c \in \mathbf{Op}_0$ and $[\![f]\!] \in \bigcap_{\sigma\to\tau \text{s.t.} f:\sigma\to\tau} [\![\sigma]\!] \to [\![\tau]\!]$ for each function $f \in \mathbf{Op}_1$. The semantics of an expression e with respect to a memory m is defined by the clauses:

$$\begin{array}{rcl} [\![x]\!]_m & = & m(x) \\ [\![c]\!]_m & = & [\![c]\!] \\ [\![f(e)]\!]_m & = & [\![f]\!]([\![e]\!]_m) \\ [\![(e_1,\ldots,e_n)]\!]_m & = & ([\![e_1]\!]_m,\ldots,[\![e_n]\!]_m) \end{array}$$

3.3 Distribution expressions

We assume given a set $\mathcal{D}$ of base distributions and define the set of distributions

Definition 3.6 (Distribution expressions)**.** The set **DExpr** of distribution expressions is defined by the following syntax:

$$\begin{array}{rll} d ::= | & \mathsf{bern}_p & \text{Bernoulli distribution} \\ | & \mathcal{U}_X & \text{uniform distribution} \\ | & d_1 \times \ldots \times d_n & \text{product distribution} \end{array}$$

where $p \in [0,1]$ and X is a constant finite set or finite type.

Typical examples of uniform distributions are sampling from a bit b, sampling uniformly over integers modulo q (i.e. elements of $\mathbb{Z}_q$), non-zero integers modulo q (i.e. elements of $\mathbb{Z}_q^*$), bitstrings of length ℓ.

Each distribution expression d has a distribution type of the form $\mathbb{D}(\sigma)$ where σ is a type. Moreover, we assume that every distribution expression d is intrepreted as a full distribution over the interpretation of its type. Note that, since all distributions are constant, the semantics $[\![d]\!]$ of a distribution expression is not parametrized by a memory.

Remark 3.2. In contrast to Barthe *et al.* (2009), we only consider probabilistic samplings over full, constant, distributions. This restriction greatly simplifies the exposition of the next chapters, notably for defining a termination analysis and proof rules of the program logics, and is satisfied by all standard examples from the cryptography literature.

3.4 Statements

Statements are built from deterministic assignments, probabilistic assignments, conditionals, loops, sequencing, and procedure calls. For simplicity of exposition, we require that procedure calls are not recursive.

Definition 3.7 (Statements)**.** Let **Proc** be a set of procedure names. The set **Cmd** of statements is defined by the following syntax:

$c ::=$	abort	abort
	$\mid \mathsf{skip}$	noop
	$\mid x \leftarrow e$	deterministic assignment
	$\mid x \xleftarrow{\$} d$	probabilistic assignment
	$\mid c; c$	sequencing
	$\mid \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c$	conditional
	$\mid \mathsf{while}\ e\ \mathsf{do}\ c$	while loop
	$\mid x \leftarrow \mathcal{F}(e)$	procedure call

Notation 3.1. When there is no risk of confusion, we write $x \xleftarrow{\$} X$ instead of $x \xleftarrow{\$} \mathcal{U}_X$.

A program is a list of procedure declarations, which fix the local variables, including formal parameters, the body and the return expression of procedures.

Definition 3.8 (Program)**.** A program $\mathfrak{P}$ is given by a set **Proc** of procedures, and for every $\mathcal{F} \in \mathbf{Proc}$ a declaration of the form $\mathcal{F}(x) = [X]c; \mathsf{return}\ e$, where x is a (tuple of) local variable(s), X is a set of local variables such that $x \in X$, c is a statement called the body of the procedure, and e is a return expression. Each program must have a distinguished procedure, called its main procedure.

We let $\mathbf{args}_{\mathcal{F}}$, $\mathbf{vars}_{\mathcal{F}}$, $\mathbf{body}_{\mathcal{F}}$ and $\mathbf{res}_{\mathcal{F}}$ denote the formal parameter, local variables, body and return expression of the procedure $\mathcal{F}$.

Programs are subject to well-formedness criteria: procedures should only use local variables in their scope, and not perform recursive calls. Well-formedness can be enforced with a simple proof system, that takes as input a well-founded order on procedure names, with the main

procedure as its top elements and checks (among other things) that calls are performed on smaller procedures with respect to this order.

Remark 3.3. The requirement that each procedure definition contains exactly one return instruction mildly simplifies the definition of the program semantics. Moreover, the requirement is without loss of generality: statements with multiple return instructions can be transformed into statements with a single return instruction. Nevertheless, we will relax this requirement in examples, to improve the readability of games.

Statements are equipped with a type system, which ensures that expressions and distributions are assigned to variables of compatible types and that guards of conditionals and loops are booleans. The typing rules are straightforward (Figure 3.1).

We now turn to give a denotational semantics to statements.

Definition 3.9 (Semantics of statements)**.** The denotational semantics $[\![s]\!]$ of a statement s is a function that assigns to every memory $m \in \mathbf{Mem}$ a sub-distribution $[\![s]\!]_m \in \mathbb{D}(\mathbf{Mem})$. The definition of $[\![s]\!]_m$ is given in Figure 3.2.

The semantics of abort is the constant function that maps every initial memory to the null sub-distribution, and the semantics of skip is the function that maps every memory m to the Dirac distribution $\mathbb{1}_m$.

The semantics of a deterministic assignment is a map that takes as input an inital memory m and returns the Dirac distribution $\mathbb{1}_{m[x \leftarrow v]}$, where $m[x \leftarrow v]$ is the memory obtained by updating m with the value v resulting from the evaluation e in memory m.

The semantics of a probabilistic assignment is defined in a similar way. Concretely, the semantics of a probabilistic assignment is a map that takes as input an inital memory m, evaluates the distribution expression d in m, samples v from the resulting distribution and returns the Dirac distribution $\mathbb{1}_{m[x \leftarrow v]}$.

The semantics of a sequential composition is defined as the monadic composition of the semantics of the first and second statements.

The semantics of conditional statements is straightforward: given a memory m, one evaluates the guard e of the conditional in m, and

$$\frac{}{\vdash \mathsf{abort}}\ [\textsc{Abort}]$$

$$\frac{}{\vdash \mathsf{skip}}\ [\textsc{Skip}]$$

$$\frac{\vdash x : \sigma \qquad \vdash e : \sigma}{\vdash x \leftarrow e}\ [\textsc{Ass}]$$

$$\frac{\vdash x : \sigma \qquad \vdash d : \mathbb{D}(\sigma)}{\vdash x \xleftarrow{\$} d}\ [\textsc{Rand}]$$

$$\frac{\vdash c_1 \qquad \vdash c_2}{\vdash c_1; c_2}\ [\textsc{Seq}]$$

$$\frac{\vdash e : \{0,1\} \qquad \vdash c_1 \qquad \vdash c_2}{\vdash \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2}\ [\textsc{Seq}]$$

$$\frac{\vdash e : \{0,1\} \qquad \vdash c}{\vdash \mathsf{while}\ e\ \mathsf{do}\ c}\ [\textsc{While}]$$

$$\frac{\begin{array}{c}\vdash \mathbf{body}_{\mathcal{F}} \\ \vdash \mathbf{res}_{\mathcal{F}} : \sigma \qquad \vdash \mathbf{args}_{\mathcal{F}} : \tau \qquad \vdash e : \tau \qquad \vdash x : \sigma\end{array}}{\vdash x \leftarrow \mathcal{F}(e)}\ [\textsc{Call}]$$

Figure 3.1: Typing rules for statements

return the output sub-distribution $[\![c_1]\!]_m$ of the true branch if the guard evaluates to true and the output sub-distribution $[\![c_2]\!]_m$ of the false branch if the guard evaluates to false.

The semantics of while loops relies on the existence of limit distributions.

Proposition 3.1. Let $(\mu_i)_{i\in\mathbb{N}} \in \mathbb{D}(A)$ be an increasing family of sub-distributions, i.e.

$$\mathbb{P}_{x\sim\mu_i}[x = a] \leq \mathbb{P}_{x\sim\mu_{i+1}}[x = a]$$

for every $a \in A$ and $n \geq 1$. In particular, it follows that the sequence $\mathbb{P}_{x\sim\mu_i}[x = a]$ has a limit in $[0, 1]$ for every $a \in A$. The limit distribution of $(\mu_i)_i$, written $\lim_{i\to\infty} \mu_i$, is defined by the clause:

$$\mathbb{P}_{x\sim\lim_{i\to\infty}\mu_i}[x = a] = \lim_{i\to\infty} \mathbb{P}_{x\sim\mu_i}[x = a]$$

for every $a \in A$.

The semantics of a loop while e do c is defined as the limit of its lower approximations, which will be defined by the above proposition. For every $i \in \mathbb{N}$, the i-th approximation of while e do c is defined as the $[\![(\text{if } e \text{ then } c)^i]\!]$, where c^n is the i-fold sequential composition of c. In effect, the i-th approximation is equal to the i-fold monadic composition of $[\![\text{if } e \text{ then } c]\!]$. One cannot apply Proposition 3.1 to prove the existence of the limit of approximations, because they do not form an increasing family. However, one can define the i-th lower approximation of while e do c, by truncating from the i-th approximation all memories that satisfy e, and on which the loop must be iterated further. Formally, the i-th lower iteration of the loop while e do c is defined as

$$\lambda m.\, \mathbb{E}_{\xi\sim[\![(\text{if } e \text{ then } c)^i]\!]_m}[[\![\text{if } e \text{ then abort}]\!]_\xi]$$

This yields an increasing sequence of sub-distributions, and hence its limit exists by Proposition 3.1. Although we do not use it in the semantics, or in order to justify the soundness of proof rules, it is interesting to note that whenever $|[\![\text{while } e \text{ do } c]\!]_m| = 1$, the limit of approximations exists, and coincides with the limit of lower approximations. The proof of this fact is based on the observation that lower approximations are

below approximations, so that the limit of their weight is equal to 1, and on Theorem 1.1.

The semantics of a procedure call is defined in two steps. First, we define the semantics of a procedure $\mathcal{F}$ as a function that takes as input an initial memory and an initial value corresponding to the evaluation of the parameter, computes the semantics of the loop body on the memory $(m_{\mathrm{glob}}, m_{\mathrm{def}(\mathcal{F})[\mathbf{arg}_F \leftarrow v]})$, yielding a memory m', and returns the memory $((m'_{\mathrm{glob}}, m_{\mathrm{loc}})$ in which global variables have been updated and local variables are restored to their original value, together with the evaluation of the return expression e in memory m'. Procedure bodies may themselves contain procedure calls, but the semantics is well-defined on well-formed programs, because there exists a well-founded order on procedures, and the calling relationship must respect this order.

$$
\begin{aligned}
[\![\mathsf{abort}]\!]_m &= \mathbb{0} \\
[\![\mathsf{skip}]\!]_m &= \mathbb{1}_m \\
[\![x \leftarrow e]\!]_m &= \mathbb{1}_{m[x \leftarrow [\![e]\!]_m]} \\
[\![x \xleftarrow{\$} d]\!]_m &= \mathbb{E}_{v \sim [\![d]\!]_m}[\mathbb{1}_{m[x \leftarrow v]}] \\
[\![c_1; c_2]\!]_m &= \mathbb{E}_{\xi \sim [\![c_1]\!](m)}[[\![c_2]\!](\xi)] \\
[\![\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!]_m &= \begin{cases} [\![c_1]\!]_m & \text{if } [\![e]\!]_m = \top \\ [\![c_2]\!]_m & \text{if } [\![e]\!]_m = \bot \end{cases} \\
[\![\mathsf{while}\ e\ \mathsf{do}\ c]\!]_m &= \sup_{i \in \mathbb{N}} \left(\mathbb{E}_{\xi \sim [\![(\mathsf{if}\ e\ \mathsf{then}\ c)^i]\!]_m}[[\![\mathsf{if}\ e\ \mathsf{then}\ \mathsf{abort}]\!]_\xi] \right) \\
[\![x \leftarrow \mathcal{F}(e)]\!]_m &= \mathbb{E}_{(m', v') \sim [\![F]\!]_{(m, [\![e]\!]_m)}}[\mathbb{1}_{m'[x \leftarrow v']}] \\
[\![\mathcal{F}]\!]_{(m,v)} &= \mathsf{let}\ m_0 = (m_{\mathrm{glob}}, m_{\mathrm{def}(\mathcal{F})[\mathbf{arg}_{\mathcal{F}} \leftarrow v]})\ \mathsf{in} \\
&\qquad \mathbb{E}_{m' \sim [\![\mathbf{body}_{\mathcal{F}}]\!]_{m_0}}[\mathbb{1}_{((m'_{\mathrm{glob}}, m_{\mathrm{loc}}), [\![e]\!]_{m'})}]
\end{aligned}
$$

Figure 3.2: Denotational semantics of statements

Remark 3.4. The semantics of probabilistic programs has been studied extensively, generally in the more broader context of continuous

distributions and often for programs that combine probabilities and non-determinism. A landmark work of Kozen (1981) uses Banach fixpoint theorem and measure-theoretical tools to give a denotational semantics for a purely probabilistic language similar to ours. The probabilistic powerdomain of Jones and Plotkin (1989) is the canonical formalism for the denotational semantics of programs with both probabilities and non-determinism.

3.5 Termination

Probabilistic programs exhibit a rich range of termination behaviors. Almost sure termination is an important case when programs terminate with probability 1.

Definition 3.10 (Almost surely termination)**.** A statement c is almost surely terminating, written $\mathsf{ast}(c)$, if $|[\![c]\!]_m| = 1$ for every initial memory m.

In Chapter 5, we define a logic that can be used to prove almost sure termination of program with loops. However, it is already easy to see that every loop-free program is almost surely terminating; in fact, it satisfies a stronger property, called certain termination, i.e. there exists $n \in \mathbb{N}$ so that the program completes its execution in less than n steps on arbitrary inputs.

3.6 Exercises

1. Show that the semantics of a statement is indeed a discrete sub-distribution, i.e. its support is discrete.
2. Show sampling from sub-distributions can always be simulated
3. Show abort has same semantics as while true do skip

4

Probabilistic Relational Hoare Logic

Our central tool for reasoning about the relationship between two games is probabilistic Relational Hoare Logic (Barthe *et al.*, 2009), a relational program logic that can be used to prove the existence of probabilistic couplings between the output distributions of two programs.

4.1 Relational assertions

Relational assertions are first-order formulae whose interpretation is taken over two memories. Therefore, basic relational assertions are of the form $P(t_1, \ldots, t_n)$ where the predicate P is specified in the underlying theory, and the relational expressions $t_1, \ldots, t_n$ are built from function symbols of the underlying theory, logical variables and tagged variables of the form $x\langle 1\rangle$ and $x\langle 2\rangle$ where x is a program variable. Here the tags $\langle 1\rangle$ and $\langle 2\rangle$ are used to indicate that the interpretation of x should be taken in the first and second memory respectively. Therefore, in particular, the relational assertion $x\langle 1\rangle = x\langle 2\rangle$ captures the fact that the value of x in the left memory is equal to the value of x in the right memory.

The interpretation of a relational assertion Φ in a pair of memories

(m_1, m_2) is a boolean value $[\![\Phi]\!]_{(m_1,m_2)}$. We also define the relation $[\![\Phi]\!] \subseteq \mathbf{Mem} \times \mathbf{Mem}$ consisting of all the set of pairs of memories (m_1, m_2) such that $[\![\Phi]\!]_{(m_1,m_2)}$ holds. The definition is standard (modulo the use of two memories to interpret tagged variables) and ommitted. Note that every relational assertion Φ depends on a set of variables vars(Φ) such that for every (m_1, m_2) and (m_1', m_2'), we have $(m_1, m_2) = (m_1', m_2')$ implies $[\![\Phi]\!]_{(m_1,m_2)}$ iff $[\![\Phi]\!]_{(m_1',m_2')}$. The definition of vars is standard and ommitted.

We also consider assertions; as usual, these are first-order formulae built from program variables. The interpretation of an assertion ϕ in a memory m is a boolean value $[\![\phi]\!]_m$. memories (m_1, m_2) is a boolean value $[\![\phi]\!]_{(m_1,m_2)}$. We let $[\![\phi]\!] \subseteq \mathbf{Mem}$ denote the set of memories m such that $[\![\phi]\!]_m$ holds. Every assertion ϕ yields two relational assertions $\phi\langle 1\rangle$ and $\phi\langle 2\rangle$, with the expected relational interpretation; $[\![\phi\langle 1\rangle]\!]_{(m_1,m_2)} = [\![\phi]\!]_{m_1}$ and $[\![\phi\langle 2\rangle]\!]_{(m_1,m_2)} = [\![\phi]\!]_{m_2}$.

Notation 4.1. For every expression e, we let $e\langle 1\rangle$ and $e\langle 2\rangle$ denote the generalized expressions obtained by tagging every variable in e with a $\langle 1\rangle$ and $\langle 2\rangle$ respectively. For instance, if e is $x + y$ then $e\langle 1\rangle$ is defined as $x\langle 1\rangle + y\langle 1\rangle$.

4.2 Judgments

Judgments of probabilistic relational Hoare logic are of the form

$$\models c_1 \sim c_2 : \Phi \Rightarrow \Psi$$

and relate two programs, c_1 and c_2, w.r.t. a pre-condition Φ and a post-condition Ψ. Informally, a judgment is valid if for every pair of initial memories m_1 and m_2 related by the pre-condition Φ, there exists a Ψ-coupling that relates the sub-distributions $[\![c_1]\!]_{m_1}$ and $[\![c_2]\!]_{m_2}$.

Definition 4.1 (Valid judgment)**.** The judgment $\models c_1 \sim c_2 : \Phi \Rightarrow \Psi$ is valid if $([\![c_1]\!]_{m_1}, [\![c_2]\!]_{m_2}) \in [\![\Psi]\!]^\sharp$ for every pair of memories (m_1, m_2) such that $[\![\Phi]\!]_{(m_1,m_2)}$ holds.

Asserting the existence of a coupling, rather than constructing it explicitly, is sufficient for the class of applications considered in this

monograph. One can however extend the notion of valid judgment and the proof system to make the coupling explicit, in the form of a product program (Barthe *et al.*, 2017).

4.3 Probabilistic inequalities

The security of cryptographic constructions is ultimately captured by probabilistic inequalities (quantified over adversaries). Fortunately, pRHL judgments can be used to derive probabilistic (in)equalities, when their post-conditions have an adequate form. Concretely, the next lemmas are immediate consequences of the results from Chapter 1.

The following lemma establishes an inequality between the probability of two events in two different games and is useful, for instance, in reduction steps.

Lemma 4.1. If $\models c_1 \sim c_2 : \Phi \Rightarrow \phi_1\langle 1\rangle \Longrightarrow \phi_2\langle 2\rangle$ then for every two memories m_1 and m_2 such that $[\![\Phi]\!]_{(m_1,m_2)}$ holds, we have:

$$\mathbb{P}_{[\![c_1]\!]_{m_1}}[[\![\phi_1]\!]] \leq \mathbb{P}_{[\![c_2]\!]_{m_2}}[[\![\phi_2]\!]]$$

The next lemma establishes an upper bound between the difference of probability of two events in two different games and is useful in failure event steps.

Lemma 4.2. If $\models c_1 \sim c_2 : \Phi \Rightarrow \phi_1\langle 1\rangle \Longrightarrow (\phi_2\langle 2\rangle \vee F\langle 2\rangle)$ then for every two memories m_1 and m_2 such that $[\![\Phi]\!]_{(m_1,m_2)}$ holds, we have

$$\mathbb{P}_{[\![c_1]\!]_{m_1}}[[\![\phi_1]\!]] - \mathbb{P}_{[\![c_2]\!]_{m_2}}[[\![\phi_2]\!]] \leq \mathbb{P}_{[\![c_2]\!]_{m_2}}[[\![F]\!]]$$

The previous lemma also has a symmetric version.

Lemma 4.3. If $\models c_1 \sim c_2 : \Phi \Rightarrow (\phi_1\langle 1\rangle \wedge F_1\langle 1\rangle) \Longleftrightarrow (\phi_2\langle 2\rangle \wedge F_2\langle 2\rangle)$ then for every two memories m_1 and m_2 such that $[\![\Phi]\!]_{(m_1,m_2)}$ holds, we have

$$\mathbb{P}_{[\![c_1]\!]_{m_1}}[[\![\phi_1]\!]] - \mathbb{P}_{[\![c_2]\!]_{m_2}}[[\![\phi_2]\!]] \leq \max(\mathbb{P}_{[\![c_1]\!]_{m_1}}[[\![\neg F_1]\!]], \mathbb{P}_{[\![c_2]\!]_{m_2}}[[\![\neg F_2]\!]])$$

The next lemma establishes that an event has the same probability in two different games and is useful in bridging steps.

Lemma 4.4. If $\models c_1 \sim c_2 : \Phi \Rightarrow \bigwedge_{x\in\mathcal{X}} x\langle 1\rangle = x\langle 2\rangle$ then for every two memories m_1 and m_2 such that $[\![\Phi]\!]_{(m_1,m_2)}$ holds, and for every event ϕ that only depends on $\mathcal{X}$, we have

$$\mathbb{P}_{[\![c_1]\!]_{m_1}}[[\![\phi]\!]] = \mathbb{P}_{[\![c_2]\!]_{m_2}}[[\![\phi]\!]]$$

The next lemma establishes an upper bound on the difference of probability of the same event in two different games, and is useful for failure event steps.

Lemma 4.5. If $\models c_1 \sim c_2 : \Phi \Rightarrow F\langle 2\rangle \rightarrow \bigwedge_{x\in\mathcal{X}} x\langle 1\rangle = x\langle 2\rangle$ then for every two memories m_1 and m_2 such that $[\![\Phi]\!]_{(m_1,m_2)}$ holds, and for every event ϕ that only depends on $\mathcal{X}$, we have

$$\mathbb{P}_{[\![c_1]\!]_{m_1}}[[\![\phi]\!]] - \mathbb{P}_{[\![c_2]\!]_{m_2}}[[\![\phi]\!]] \leq \mathbb{P}_{[\![c_2]\!]_{m_2}}[[\![F]\!]]$$

4.4 Proof system

We now present a proof system for deriving valid pRHL judgments. The proof rules are split into three groups:

structural rules: these rules can be applied independently of the shape of the programs;

2-sided rules: these rules requires that the two programs have a specific and corresponding shape (for instance, the two programs must be a deterministic assignment; or the two programs must be a conditional statement). There is one single rule for each form of statement;

1-sided rules: these rules requires that one of the two programs has a specific shape. There are two rules per for each form of statement; a left rule requiring that the left statement of the judgment has the desired shape, and a right rule requiring that the right statement has the desired shape.

4.4.1 Structural rules

The [False] rule states that arbitrary programs are related, when the precondition is provably false.

The [Conseq] rule is similar to the rule of consequence in Hoare Logic, and can be used for weakening the post-condition and strengthening the pre-condition.

The [Frame] rule allows to strengthen simultaneously the pre-condition and the post-condition of a valid judgment with an assertion Θ, provided the variables modified by the two statements of the judgment do not appear in Θ. The formal definition of modified variables is given below.

Definition 4.2 (Modified variables)**.** The set $\operatorname{mod} c$ of modified variables of a statement c by the clauses:

$$\begin{aligned}
\operatorname{mod}(x \leftarrow e) &= \{x\} \\
\operatorname{mod}(x \xleftarrow{\$} d) &= \{x\} \\
\operatorname{mod}(c_1; c_2) &= \operatorname{mod}(c_1) \cup \operatorname{mod}(c_2) \\
\operatorname{mod}(\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) &= \operatorname{mod}(c_1) \cup \operatorname{mod}(c_2) \\
\operatorname{mod}(\mathsf{while}\ e\ \mathsf{do}\ c) &= \operatorname{mod}(c) \\
\operatorname{mod}(x \leftarrow f(e)) &= \{x\} \cup \operatorname{mod}_{\mathrm{glob}}(\mathbf{body}_f)
\end{aligned}$$

where $\operatorname{mod}_{\mathrm{glob}}(c) \triangleq \operatorname{mod}(c) \cap \mathbf{Vars}_{\mathrm{glob}}$.

The [Case] rule allows proving a judgment by case analysis; specifically, the validity of a judgment with pre-condition $\Phi_1 \vee \Phi_2$ can be established from the validity of two judgments, one where the pre-condition is Φ_1 and another one where the precondition is Φ_2. The rule is typically used for performing a case distinction on the value of a boolean expression e; in this case, the original precondition is strengthening with e in the first judgment and with $\neg e$ in the second judgment.

The [Exists] rule is similar to the [Case] rule, excepts that it considers the case where the pre-condition is an existential statement.

The [Struct] rule allows replacing programs by provably equivalent programs. The rule depends on an auxiliary judgment of the form $\Phi \models c \equiv c'$, where Φ is a relational assertion, and which states that c and c' are equivalent (i.e. yield equal distributions) for every two pairs of memories that satisfy Φ. We leave the proof system for program equivalence unspecified.

$$\frac{}{\models c_1 \sim c_2 : \bot \Rightarrow \Psi}\ [\textsc{False}]$$

$$\frac{\models c_1 \sim c_2 : \Phi' \Rightarrow \Psi' \qquad \Phi \Longrightarrow \Phi' \qquad \Psi' \Longrightarrow \Psi}{\models c_1 \sim c_2 : \Phi \Rightarrow \Psi}\ [\textsc{Conseq}]$$

$$\frac{\models c_1 \sim c_2 : \Phi \Rightarrow \Psi \quad \mathrm{vars}(\Theta) \cap (\mathrm{mod}(c_1)\langle 1\rangle \cup \mathrm{mod}(c_2)\langle 2\rangle) = \emptyset}{\models c_1 \sim c_2 : \Phi \wedge \Theta \Rightarrow \Psi \wedge \Theta}\ [\textsc{Frame}]$$

$$\frac{\models c_1 \sim c_2 : \Phi_1 \Rightarrow \Psi \quad \models c_1 \sim c_2 : \Phi_2 \Rightarrow \Psi}{\models c_1 \sim c_2 : \Phi_1 \vee \Phi_2 \Rightarrow \Psi}\ [\textsc{Case}]$$

$$\frac{\forall x : T.\ \models c_1 \sim c_2 : \Phi \Rightarrow \Psi}{\models c_1 \sim c_2 : \exists x : T.\, \Phi \Rightarrow \Psi}\ [\textsc{Exists}]$$

$$\frac{\models c_1 \sim c_2 : \Phi \Rightarrow \Psi \qquad \Phi \models c_1 \equiv c_1' \qquad \Phi \models c_2 \equiv c_2'}{\models c_1' \sim c_2' : \Phi \Rightarrow \Psi}\ [\textsc{Struct}]$$

Figure 4.1: Structural rules

Remark 4.1. Not all intuitive structural rules are sound. In particular, the following rule is not sound, because liftings are not closed under conjunction (see Proposition 1.1).

$$\frac{\models c_1 \sim c_2 : \Phi \Rightarrow \Psi_1 \quad \models c_1 \sim c_2 : \Phi \Rightarrow \Psi_2}{\models c_1 \sim c_2 : \Phi \Rightarrow \Psi_1 \wedge \Psi_2}$$

4.4.2 Two-sided rules

The [Assg] rule states that an assertion is valid after two assignments, if the original pairs of memories satisfies the assertion obtained by substituting in place of the variables being assigned the (tagged) expressions of the assignments.

The [Rand] rule requires that there exists a bijective coupling between the two distributions of the left and right programs. Moreover, in order for Ψ to be a valid post-condition, the initial memories should satisfy that $\Psi[v/x_1\langle 1\rangle][h(v)/x_2\langle 1\rangle]$ for every $v \in T_1$

The [Seq] rule for sequential composition simply reflects the compositional property of couplings.

The [Cond] rule considers two conditional statements that execute in lockstep. Specifically, it requires that the pre-condition Φ implies that the guards of the two conditional statements are logically equivalent. The premises of the rule ensure that that both the true branches of the statements and the false branches of the statements are related by pre-condition Φ (strengthened by the guard of the conditionals or their negation) and the same post-condition Ψ. Therefore the two conditional statements are related by the pre-condition Φ and the post-condition Ψ.

The [While] rule considers two while loops that execute in lockstep. Specifically, it requires that there exists a loop invariant Θ that is initially valid and preserved by one iteration of the two loop bodies, and such that the loop guards are equivalent for any two memories satisfying the invariant. Upon termination, i.e. in the output distributions, both loop guards are false and the loop invariant is valid.

The [Call] rule considers the case where the left and right statements perform a procedure call, possibly with different procedures. It

$$\models \mathsf{skip} \sim \mathsf{skip} : \Psi \Rightarrow \Psi \ [\textsc{Skip}]$$

$$\frac{\models c_1 \sim c_2 : \Phi \Rightarrow \Theta \quad \models c_1' \sim c_2' : \Theta \Rightarrow \Psi}{\models c_1; c_1' \sim c_2; c_2' : \Phi \Rightarrow \Psi} \ [\textsc{Seq}]$$

$$\frac{}{\models x_1 \leftarrow e_1 \sim x_2 \leftarrow e_2 : \Psi[e_1\langle 1\rangle / x_1\langle 1\rangle][e_2\langle 2\rangle / x_2\langle 2\rangle] \Rightarrow \Psi} \ [\textsc{Assn}]$$

$$\frac{\begin{array}{c} h \blacktriangleleft \langle [\![\mu_1]\!] \,\&\, [\![\mu_2]\!] \rangle \\ \Phi \triangleq \forall v : T_1, v \in \mathrm{supp}(\mu_1) \implies \Psi[v/x_1\langle 1\rangle][h(v)/x_2\langle 2\rangle] \end{array}}{\models x_1 \xleftarrow{\$} d_1 \sim x_2 \xleftarrow{\$} d_2 : \Phi \Rightarrow \Psi} \ [\textsc{Rand}]$$

$$\frac{\begin{array}{c} \Phi \implies e_1\langle 1\rangle = e_2\langle 2\rangle \\ \models c_1 \sim c_2 : \Phi \wedge e_1\langle 1\rangle \Rightarrow \Psi \qquad \models c_1' \sim c_2' : \Phi \wedge \neg e_1\langle 1\rangle \Rightarrow \Psi \end{array}}{\models \mathsf{if}\ e_1\ \mathsf{then}\ c_1\ \mathsf{else}\ c_1' \sim \mathsf{if}\ e_2\ \mathsf{then}\ c_2\ \mathsf{else}\ c_2' : \Phi \Rightarrow \Psi} \ [\textsc{Cond}]$$

$$\frac{\Theta \implies e_1\langle 1\rangle = e_2\langle 2\rangle \qquad \models c_1 \sim c_2 : \Theta \wedge e_1\langle 1\rangle \Rightarrow \Theta}{\models \mathsf{while}\ e_1\ \mathsf{do}\ c_1 \sim \mathsf{while}\ e_2\ \mathsf{do}\ c_2 : \Theta \Rightarrow \Theta \wedge \neg e_1\langle 1\rangle} \ [\textsc{While}]$$

$$\frac{\begin{array}{c} \Phi' \triangleq \Phi[e_1\langle 1\rangle / \mathbf{args}_{\mathcal{F}_1}\langle 1\rangle][e_2\langle 2\rangle / \mathbf{args}_{\mathcal{F}_2}\langle 2\rangle] \\ \models \mathbf{body}_{\mathcal{F}_1} \sim \mathbf{body}_{\mathcal{F}_2} : \Phi' \Rightarrow \Psi[\mathbf{res}_{\mathcal{F}_1}\langle 1\rangle / x_1\langle 1\rangle][\mathbf{res}_{\mathcal{F}_2}\langle 2\rangle / x_2\langle 2\rangle] \end{array}}{\models x_1 \leftarrow \mathcal{F}_1(e_1) \sim x_2 \leftarrow \mathcal{F}_2(e_2) : \Phi' \Rightarrow \Psi} \ [\textsc{Call}]$$

Figure 4.2: Two-sided rules

requires to establish a relation between the bodies of the two procedures, under a strengthened pre-condition that sets the values of the formal parameters to be equal to the arguments e_1 and e_2 of the procedure calls, respectively.

4.4.3 One-sided rules

We only present left rules (right rules are similar). In all cases, except for the rule for conditionals, the program on the right is a skip statement.

The [Assg-L] rule states that an assertion Ψ is a valid post-condition,

if the initial pair of memories satisfy the assertion $\Psi[e\langle 2\rangle / x\langle 1\rangle]$.

The [Rand-L] rule states that an assertion Ψ is a valid post-condition, if the initial pair of memories satisfy the assertion $\forall v \in \mathrm{supp}(d_1).\ \Psi[v/x\langle 1\rangle]$. In this case, the rule treats random assignments as a non-deterministic assignment, as there is no opportunity to couple the random assignment on the left with a random assignment on the right.

The [Cond-L] rule considers a conditional statement on the left and an arbitrary statement to the right. It performs a case analysis on the guard of the conditional statement and matches its true and false branch against the right statement.

The [While-L] rule requires each iteration of the loop body preserves an invariant Θ, and that the loop is almost surely terminating. If the initial pair of memories satisfy Θ, then upon termination, i.e. in the output distributions, the loop guard is false and the loop invariant is valid.

The [Call-L] considers the case where the left statement performs a procedure call, and where the right statement is arbitrary. The rule requires to establish a relation between the body of the procedure and the right statement, under a strengthened pre-condition that the value of the formal parameters is equal to the argument e_1 of the procedure call.

Note that there is no one-sided rule for sequential composition.

Remark 4.2. Barthe *et al.* (2017) prove that, given a sufficiently strong proof system for structural equivalence on programs, all one-sided rules except [While-L] can be derived from their two-sided counterpart.

4.4.4 Soundness and completeness

The proof system is *sound*, in the sense that the conclusions of all the proof rules are valid judgments, provided the premises of the rules are valid judgments and the side-conditions, if any, hold. On the other hand, the proof system is *incomplete*, in the sense that there are valid judgments that may not be derived from the proof system.

One source of incompleteness is the rule for loops, which requires that the two loops make the same number of iterations. It is possible

$$\frac{}{\models x_1 \leftarrow e_1 \sim \mathsf{skip} : \Psi[e_1\langle 1\rangle / x_1\langle 1\rangle] \Rightarrow \Psi} \text{ [Assg-L]}$$

$$\frac{}{\models x_1 \xleftarrow{\$} d_1 \sim \mathsf{skip} : \forall v_1 \in \mathrm{supp}(d_1), \Psi[v/x_1\langle 1\rangle] \Rightarrow \Psi} \text{ [Rand-L]}$$

$$\frac{\models c_1 \sim c_2 : \Phi \wedge e_1\langle 1\rangle \Rightarrow \Psi \qquad \models c_1' \sim c_2 : \Phi \wedge \neg e_1\langle 1\rangle \Rightarrow \Psi}{\models \mathsf{if}\ e_1\ \mathsf{then}\ c_1\ \mathsf{else}\ c_1' \sim c_2 : \Phi \Rightarrow \Psi} \text{ [Cond-L]}$$

$$\frac{\models c_1 \sim \mathsf{skip} : \Theta \wedge e_1\langle 1\rangle \Rightarrow \Theta \qquad \mathsf{ast}(\mathsf{while}\ e_1\ \mathsf{do}\ c_1)}{\models \mathsf{while}\ e_1\ \mathsf{do}\ c_1 \sim \mathsf{skip} : \Theta \Rightarrow \Theta \wedge \neg e_1\langle 1\rangle} \text{ [While-L]}$$

$$\frac{\begin{array}{c}\Phi' \triangleq \Phi[e_1\langle 1\rangle / \mathbf{args}_{f_1}\langle 1\rangle] \\ \models \mathbf{body}_{f_1} \sim \mathsf{skip} : \Phi' \Rightarrow \Psi[x_1\langle 1\rangle / \mathbf{res}_{f_1}\langle 1\rangle]\end{array}}{\models x_1 \leftarrow f_1(e_1) \sim \mathsf{skip} : \Phi' \Rightarrow \Psi} \text{ [Call-L]}$$

Figure 4.3: One-sided (left) rules

to palliate for this incompleteness within the proof system described above in two different ways: first, by using the one-sided rules for loops; however, this is possible when the two programs have extremely well-behaved termination behavior; second, by using the [Struct] rule to perform standard loop optimizations that make structurally different loops amenable to verification. Another solution, outside of the proof system described above, is to use a more general rule for loops that does not require the two programs to make the same number of iterations. We only present a simplified rule, and refer the reader to (Barthe *et al.*,

2017) for the most general rule:

$$\frac{\begin{array}{c}\Theta \implies (e_1 \vee e_2) = e \\ \Theta \wedge e \implies \oplus\{p_0, p_1, p_2\} \\ \Theta \wedge p_0 \wedge e \implies e_1 = e_2 \\ \Theta \wedge p_1 \wedge e \implies e_1 \\ \Theta \wedge p_2 \wedge e \implies e_2 \\ |\mathsf{while}\ (e_1 \wedge p_1)\ \mathsf{do}\ c_1| = |\mathsf{while}\ (e_2 \wedge p_2)\ \mathsf{do}\ c_2| = 1 \\ \models \mathsf{if}\ e_1\ \mathsf{then}\ c_1 \sim \mathsf{if}\ c_2\ \mathsf{then}\ e_2 : \Theta \wedge p_0 \Rightarrow \Theta \\ \models c_1 \sim \mathsf{skip} : \Theta \wedge e_1 \wedge p_1 \Rightarrow \Theta \\ \models \mathsf{skip} \sim c_2 : \Theta \wedge e_2 \wedge p_2 \Rightarrow \Theta\end{array}}{\models \mathsf{while}\ e_1\ \mathsf{do}\ c_1 \sim \mathsf{while}\ e_2\ \mathsf{do}\ c_2 : \Theta \Rightarrow \Theta \wedge \neg e_1 \wedge \neg e_2}$$

The rule interleaves synchronous and asynchronous executions of the loop bodies, as reflected by its last three premises. The first set of premises defines the conditions under which interleavings must be considered. The first premise specifies an expression e, which may mention variables from both sides, that holds true exactly when at least one of the guards is true. Next, the next premise states that whenever e is valid, exactly one of the tests p_0, p_1, and p_2 must hold—this is captured by the notation $\oplus\{p_0, p_1, p_2\}$. These tests must satisfy some additional conditions, given in the third, fourth, and fifth premises, and guide the analysis of the loop bodies. If p_0 holds, then both guards should be equal and we can execute the two sides one iteration, preserving the loop invariant Θ. If p_1 holds and the right loop has not terminated yet, then the left loop also has not terminated yet (i.e., e_2 holds), we may execute the left loop one iteration. If p_2 holds and the left loop has not terminated yet (i.e., e_1 holds), then the right loop also has not terminated yet and we may execute the right loop one iteration. The sixth and seven premises deal with termination. Note that some condition on termination is needed for soundness of the logic: if the left loop terminates with probability 1 while the right loop terminates with probability 0 (i.e., never), it is impossible to construct a valid coupling since there is no distribution on pairs that has first marginal with weight 1 and second marginal with weight 0. So, we require that the first and second loops are a.s. terminating assuming p_1 and p_2 respectively. This ensures that with probability 1, there are only finitely many steps where

we execute the left or right loop separately.

Remark 4.3. Barthe *et al.* (2017) prove that for deterministic programs, the general rule for loops make the proof system relatively complete with respect to self-composition. More precisely, for every two deterministic and terminating programs c_1 and c_2 with disjoint sets of variables,

$$\models c_1 \sim c_2 : \Phi \Rightarrow \Psi \Leftrightarrow \models \bar{\Phi} : c_1; c_2 \Rightarrow \bar{\Psi}$$

where $\bar{\Phi}$ and $\bar{\Psi}$ are obtained from Φ and Ψ by erasing the tags of variables.

Loops are not the sole source of incompleteness; the rule for random assignments is also incomplete, for two reasons. First, the rule uses bijective couplings, which are less general than couplings. One can be lift this restriction by using a more general rule for probabilistic assignments, as proposed in (Barthe *et al.*, 2017). In our simplified setting, their rule becomes:

$$\frac{\mu \blacktriangleleft \langle [\![d_1]\!] \,\&\, [\![d_2]\!] \rangle \qquad \Phi \triangleq \forall (v_1, v_2) \in \mathrm{supp}(\mu).\ \Psi[v_1/x_1\langle 1\rangle][v_2/x_2\langle 1\rangle]}{\models x_1 \xleftarrow{\$} d_1 \sim x_2 \xleftarrow{\$} d_2 : \Phi \Rightarrow \Psi}$$

This rule is complete when considering two probabilistic assignments. However, the proof system remains incomplete, due to the interactions between probabilistic assignments and other constructions.

One source of incompleteness is the interaction between probabilistic assignments and sequential composition. In general, it might not be possible (even with the stronger rule above) to relate two sequences of assignments of different lengths, or sequences of assignments of the same length but which are performed over sets of a different size. For instance, one cannot use the rule [Rand] only to relate the two statements:

$$x \xleftarrow{\$} \{0,1\}^{\ell_1}; x \xleftarrow{\$} {}'\{0,1\}^{\ell_2}; \mathsf{return}\ (x \parallel x')$$

$$x \xleftarrow{\$} \{0,1\}^{\ell_3}; x \xleftarrow{\$} {}'\{0,1\}^{\ell_3}; \mathsf{return}\ (x \parallel x')$$

even if $\ell_1 + \ell_2 = \ell_3 + \ell_4$.

It is not clear how to establish a coupling between the two statements, even with the more general rule for probabilistic assignments. However,

one could relate the two programs using the [Struct] and [Rand] rules, given a sufficiently strong auxiliary proof system for program equivalence.

4.5 Excercises

1. Let c_1 and c_2 be statements, and let Φ and Ψ be assertions such that $x \notin \mathrm{vars}(\Phi)$ and $x \notin \mathrm{vars}(\Psi)$ and $x \notin \mathrm{vars}(e)$. Prove that the following rule is valid

$$\frac{\models \mathsf{if}\ e\ \mathsf{then}\ x \xleftarrow{\$} \mu; c_1\ \mathsf{else}\ c_2 \sim c : \Phi \Rightarrow \Psi \qquad |\mu| = 1}{\models x \xleftarrow{\$} \mu; \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \sim c : \Phi \Rightarrow \Psi}$$

   Show that the rule is unsound if $|\mu| \neq 1$.

2. Composition. Prove that the following rule is sound

$$\frac{\models c_1 \sim c_2 : \Phi \Rightarrow \Psi \qquad \models c_2 \sim c_3 : \Phi' \Rightarrow \Psi'}{\models c_1 \sim c_3 : \Phi'' \Rightarrow \Psi''}\ [\mathrm{Comp}]$$

   where $\Phi'' = \exists z_1 \ldots z_k.\Phi[x_1\langle 2\rangle, \ldots, x_k\langle 2\rangle/z_1, \ldots z_k] \wedge \Phi'[x_1\langle 1\rangle, \ldots, x_k\langle 1\rangle/z_1, \ldots$
   and Ψ is defined similarly.

3. Define a minimal set of structural rules so that all 1-sided rules are derivable from 2-sided rules.

5

Union Bound Logic

We use the Union Bound logic from Barthe *et al.* (2016b) for bounding the probability of events on output sub-distributions of probabilistic programs. The logic is based on the union bound, a very simple but effective tool from probability theory.

5.1 Judgments and validity

Judgments are of the form $\models_\beta c : \phi \Rightarrow \psi$ where c is a statement, ϕ, ψ are assertions and $\beta \in [0, 1]$ is a constant.

Informally, a judgment is valid if the probability of $\neg\psi$ in $[\![c]\!]_m$ is upper bounded by β for every memory m that satisfies the precondition ϕ.

Definition 5.1 (Valid judgment)**.** The judgment $\models_\beta c : \phi \Rightarrow \psi$ is valid if $\mathbb{P}_{[\![c]\!](m)}[\neg\psi] \leq \beta$ for every memory m such that $[\![\phi]\!]_m$ holds.

The proof rules for pHL judgments include structural rules (Figure 5.1), and rules for each construct (Figure 5.2).

We briefly discuss the rules.

The [False] rule allows us to conclude that false holds with probability at most 0 in the final memory.

$$[\textsc{False}]\ \frac{}{\models_1 c : \psi \Rightarrow \bot}$$

$$[\textsc{Conseq}]\ \frac{\models \phi' \implies \phi \qquad \models \psi \implies \psi' \qquad \beta \leq \beta' \qquad \models_\beta c : \phi \Rightarrow \psi}{\models_{\beta'} c : \phi' \Rightarrow \psi'}$$

$$[\textsc{Frame}]\ \frac{\mathrm{vars}(\psi) \cap \mathrm{mod}(c) = \emptyset}{\models_0 c : \psi \Rightarrow \psi}$$

$$[\textsc{Case}]\ \frac{\models_\beta c : \phi_1 \Rightarrow \psi \qquad \models_\beta c : \phi_2 \Rightarrow \psi}{\models_\beta c : \phi_1 \vee \phi_2 \Rightarrow \psi}$$

$$[\textsc{Exists}]\ \frac{\forall x : T.\ \models_\beta c : \phi \Rightarrow \psi}{\models_\beta c : \exists x : T.\, \phi \Rightarrow \psi}$$

$$[\textsc{And}]\ \frac{\models_{\beta_1} c : \phi \Rightarrow \psi_1 \qquad \models_{\beta_2} c : \phi \Rightarrow \psi_2}{\models_{\beta_1+\beta_2} c : \psi \Rightarrow \psi_1 \wedge \psi_2}$$

Figure 5.1: Structural pHL proof rules

$$[\textsc{Skip}]\ \frac{}{\models_0 \mathsf{skip} : \psi \Rightarrow \psi} \qquad [\textsc{Assn}]\ \frac{}{\models_0 x \leftarrow e : \psi[e/x] \Rightarrow \psi}$$

$$[\textsc{Rand}]\ \frac{\forall m.\ [\![\phi]\!]_m \implies \mathbb{P}_{[\![x \xleftarrow{\$} d]\!]_m}[\neg\psi] \leq \beta}{\models_\beta x \xleftarrow{\$} d(e) : \phi \Rightarrow \psi}$$

$$[\textsc{Seq}]\ \frac{\models_\beta c : \phi \Rightarrow \theta \qquad \models_{\beta'} c' : \theta \Rightarrow \psi}{\models_{\beta+\beta'} c; c' : \phi \Rightarrow \psi}$$

$$[\textsc{Cond}]\ \frac{\models_\beta c : \phi \wedge e \Rightarrow \psi \qquad \models_\beta c' : \phi \wedge \neg e \Rightarrow \psi}{\models_\beta \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c' : \phi \Rightarrow \psi}$$

$$[\textsc{While-CT}]\ \frac{\models_\beta c : \theta \wedge e \Rightarrow \theta \qquad \models_0 (\mathsf{if}\ e\ \mathsf{then}\ c)^n : \phi \Rightarrow \neg e}{\models_{n\beta} \mathsf{while}\ e\ \mathsf{do}\ c : \theta \wedge \phi \Rightarrow \psi \wedge \neg e}$$

$$[\textsc{While-AST}]\ \frac{\models_0 c : \theta \wedge e \Rightarrow \theta \qquad \models_\beta c : \theta \wedge e \Rightarrow \neg e \qquad \beta \neq 1}{\models_0 \mathsf{while}\ e\ \mathsf{do}\ c : \theta \Rightarrow \theta \wedge \neg e}$$

$$[\textsc{Call}]\ \frac{\phi' \triangleq \phi[e/\mathbf{args}_{\mathcal{F}}] \qquad \models_\beta \mathbf{body}_{\mathcal{F}} : \phi' \Rightarrow \psi[x/\mathbf{res}_{\mathcal{F}}]}{\models_\beta x \leftarrow \mathcal{F}(e) : \phi' \Rightarrow \psi}$$

Figure 5.2: Non-structural pHL proof rules

The [Conseq] rule allows strengthening the pre-condition, weakening the post-condition, and increasing the index—this corresponds to allowing a possibly higher probability of failure.

The frame rule [Frame] preserves assertions that do not mention variables modified by the command. The conjunction rule [And] is another instance of the union bound, allowing us to combine two post-conditions while adding up the failure probabilities. The case rule [Case] is the dual of [And] and takes the maximum failure probability among two post-conditions when taking their disjunction. Finally,

The rule for random sampling [Rand] allows us to assume a proposi-

tion ψ about the random sample provided that ψ fails with probability at most β. This is a semantic condition which we introduce as an axiom for each primitive distribution.

The remaining rules are similar to the standard Hoare logic rules, with special handling for the index. The sequence rule [Seq] states that the failure probabilities of the two commands add together; this is simply the union bound internalized in our logic. The conditional rule [If] assumes that the indices for the two branch judgments are equal—which can always be achieved via weakening—keeping the same index for the conditional. Roughly, this is because only one branch of the conditional is executed. The loop rule [While] simply accumulates the failure probability β throughout the iterations; the side conditions ensure that the loop terminates in at most k iterations except with probability $k \cdot \beta$. To reason about procedure calls, standard (internal) procedure calls use the rule [Call], which substitutes the argument and return variables in the pre- and post-condition, respectively. External procedure calls use the rule [Ext]. We do not have access to the implementation of the procedure; we know just the type of the return value.

5.2 Soundness and completeness

The logic is sound: if the premises of a proof rule are valid, and the side-conditions, if any, hold, then the conclusions of the proof rule are valid. Unsurprisingly, the proof system is incomplete: there are valid judgments that cannot be proved using the logic. Indeed, the union bound principle is a simple tool, and it induces non-optimal bounds. Concentration inequalities **dubhashi2009concentration** is an active field of research that studies advanced methods for improving these bounds, and that is not captured by pHL.

5.3 Further reading

There is a long line of research, spanning more than four decades, on program logics for reasoning about general probabilistic properties both for purely probabilistic programs and for programs that combine probabilities and non-deterministic choice. Kozen (1985) develop a

propositional dynamic logic for a purely probabilistic language; this work has later been extended in many directions. Particularly, a long line of work by McIver and Morgan, summarized in (McIver and Morgan, 2005), develops a weakest pre-expectation calculus for a language with both probabilities and non-determinism. Their work has also been extended in many directions, with applications to security and to complexity analysis.

6

Adversaries

Any formalism for reasoning about reductionist security proofs of cryptographic constructions must provide support for reasoning about adversaries. In this chapter, we introduce a generic formalization of adversaries, and show how the semantics of programs and the rules of the program logic can be extended to accommodate adversaries.

6.1 Definition

Adversarial programs are programs with two classes of procedures: program procedures, often simply called procedures, and adversarial procedures, also called adversaries.

Definition 6.1 (Adversarial program)**.** An adversarial program $\mathfrak{P}$ is given by a set **Proc** of procedures, with a distinguished main procedure, a set **AdvProc** of adversarial procedures and a list of declarations of the form $\mathcal{F}(x) = [X]c; \mathsf{return}\ e$, where x is a (tuple of) local variable(s), X is a set of local variables such that $x \in X$, c is a statement called the body of the procedure, and e is a return expression. We require that:

- there is exactly one declaration for each $\mathcal{F} \in \mathbf{Proc}$, and only the main procedure can call adversaries;

- there is at most one declaration for each $\mathcal{A} \in \mathbf{AdvProc}$, and moreover the body of $\mathcal{A}$ can call procedures in a distinguished subset $\mathcal{F}_{\mathrm{orcl}}$ of $\mathcal{F}$ or other adversaries;
- adversaries and procedures operate on two disjoint sets of variables: program variables (both global and local), and adversary variables (both global and local).

An adversarial program is *abstract* if there is no declaration for adversaries and *concrete* if there is a declaration attached to each adversary. Security notions and assumptions are expressed by abstract adversarial programs; on the other hand, concrete adversarial programs can be given a denotational semantics.

Remark 6.1. Some security notions limit the interactions between adversaries. This can be achieved by assigning to each adversary read/write permissions on adversarial variables; one can further refine these permissions to be quantitative, and to specify an upper bound on the number of reads or writes that an adversary can perform on a given variable.

Adversarial programs are naturally ordered by a refinement relation; we say that an adversarial program $\mathfrak{P}$ refines an adversarial program $\mathfrak{P}'$ if they have the same sets of procedures and every declaration in $\mathfrak{P}$ is also a declaration in $\mathfrak{P}'$.

The notion of well-formed and well-typed program extends readily to adversarial programs. For the former, we require that the call graph of an adversarial program is well-founded, and that the separation between adversary and program memory is enforced correctly.

6.2 Semantics

We give a denotational semantics of concrete adversarial programs, similarly to Chapter 3 for plain programs. The semantics is parametrized by a ressource policy, which imposes an upper bound on the number of calls that an adversary can make to a procedure.

Definition 6.2 (Resource policy)**.** A resource policy for an adversarial program $\mathfrak{P}$ is a family of natural numbers $(q_{\mathcal{F}})_{\mathcal{F}} \in \mathbf{Proc} \in \mathbb{N}$ which sets

the maximal number of calls that an adversary can make to procedure $\mathcal{F}$.

We also introduce the notion of history in order to model security definitions. A simple notion of history suffices for our purposes. However, the notion of history can be strengthened if required.

Definition 6.3 (History)**.** An history for an adversarial program $\mathfrak{P}$ is a family of list of values $(\boldsymbol{L}_{\mathcal{F}})_{\mathcal{F}} \in \mathbf{Proc} \in \mathbb{N}$ recording the list of input and output values of adversarial calls to the procedure $\mathcal{F}$.

Our semantics records the history of adversarial calls, and aborts execution whenever an adversary exceeds the maximal number of allowed calls on an oracle. Formally, the semantics of a concrete adversarial program $\mathfrak{P}$ with respect to a resource policy $(q_{\mathcal{F}})_{\mathcal{F} \in \mathbf{Proc}} \in \mathbb{N}$ is defined as a function:

$$[\![\mathfrak{P}]\!] : (\mathbf{Mem} \times \mathbf{AdvMem}) \to \mathbb{D}(\mathbf{Mem} \times \mathbf{AdvMem} \times \mathbf{Hist})_{\perp}$$

The semantics is essentially unchanged from Chapter 3, although we need to use the error monad to propagate the error $\perp$ whenever it arises in some part of the execution. The only interesting case is the semantics of calls in an adversarial procedure; well-formedness implies that these can either be oracle or adversary calls. We treat oracle calls; adversary calls are treated similarly. The semantics is defined by case analysis: if the adversary has already performed its maximal allowed number of calls, then the call returns an error. Else, the execution proceeds normally. We execute $\mathcal{F}$; since oracles do not perform adversarial calls, the adversarial memory and the history need not be updated. We return the output of executing $\mathcal{F}$, with the adversarial variable x updated with the return value of the oracle, and the history updated to record the call. In the first case, when $|\boldsymbol{L}_{\mathcal{F}}| \geq q_{\mathcal{F}}$, we set

$$[\![x \leftarrow \mathcal{F}(e)]\!]_{(m,m_a,h)} = \quad \mathsf{let}\ v = [\![e]\!]_{m_a}\ \mathsf{in} \\ \mathbb{E}_{(m',v') \sim [\![F]\!]_{(m,v)}} \left[\mathbb{1}_{(m', m_a[x \leftarrow v'], h[\boldsymbol{L}_{\mathcal{F}} \leftarrow (v,v') :: \boldsymbol{L}_{\mathcal{F}}])} \right]$$

Finally, we say that a concrete adversarial $\mathfrak{P}$ satisfies a resource policy $(q_{\mathcal{F}})_{\mathcal{F}} \in \mathbf{Proc} \in \mathbb{N}$ if for every pair of memories (m, m_a),

$$\perp \notin \mathrm{supp}([\![\mathfrak{P}]\!]_{(m,m_a)})$$

6.3 Relational reasoning about adversarial programs

The probabilistic Relational Hoare Logic from Chapter 4 extends to adversarial programs.

Assertions are similar to pRHL, and are built over adversarial and program variables. We also use a distinguished assertion $\mathsf{eqmem}_{\mathcal{A}}$ to state that the memory of an abstract adversary $\mathcal{A}$ is equal in the two executions.

A judgment $\models c_1 \sim c_2 : \Phi \Rightarrow \Psi$ is valid iff $\models c_1' \sim c_2' : \Phi \Rightarrow \Psi$ for every concrete and compatible refinements c_1' and c_2' of c_1 and c_2. Here the notion of compatibility states that c_1 and c_2 exactly have the same set of abstract adversaries, and that they get the same declaration in c_1' and c_2'. It can be shown that all the rules of pRHL retain soundness under this interpretation.

Remark 6.2. The notion of validity immediately entails the soundness of a refinement rule, which states that a judgment remains valid when replacing the left and right programs by compatible refinements. This rule offers a rudimentary mechanism for compositional reasoning, and in particular for chaining sequences of reductions and keeping the constructed adversaries implicit. However, more powerful mechanisms are required for large examples. We discuss the use of module systems for compositional reasoning in Chapter 7.

The adversary rule in pRHL is intended to compare the behavior of the same adversary in two different adversarial programs (with compatible adversaries) and reflects the fact that adversaries are purely probabilistic. Therefore, any two runs of the adversary will deliver equal outputs whenever they receive equal inputs. Moreover, adversaries do not modify the program state so every assertion which holds before calling the adversary remains valid when the adversary returns. However, adversaries may also interact with oracles, so we need to make our claim more precise. Specifically, the pRHL rule for adversaries combines two observations:

- adversaries will deliver equal queries and equal outputs whenever they are given equal inputs and equal answers to their queries;

- adversaries will preserve relational invariants on program states whenever these invariants are preserved by the oracles to which the adversary is given access.

Concretely, the adversary rule is defined as follows:

$$[\textsc{Adv}]\ \frac{\forall \mathcal{F}, y, z.\ \models y \leftarrow \mathcal{F}(z) \sim y \leftarrow \mathcal{F}(z) : z\langle 1\rangle = z\langle 2\rangle \wedge \Psi \Rightarrow y\langle 1\rangle = y\langle 2\rangle \wedge \Psi}{\models x_1 \leftarrow \mathcal{A}(e_1) \sim x_2 \leftarrow \mathcal{A}(e_2) : e\langle 1\rangle = e\langle 2\rangle \wedge \Theta \Rightarrow x\langle 1\rangle = x\langle 2\rangle \wedge \Theta}$$

where $\Theta \triangleq \Psi \wedge \mathsf{eqmem}_{\mathcal{A}}$. Note that the premise of the rule quantifies universally over all procedures; in reality, we always reason relative to a ressource policy, in which case it suffices to quantify over procedures $\mathcal{F}$ that can be called by the adversary, i.e. such that $q_{\mathcal{F}} \neq 0$.

The soundness of the adversary rule can be argumented informally by viewing an adversary as an arbitrary sequence of procedure calls; a relational assertion Ψ is a valid invariant for an arbitrary sequence of procedure calls if Ψ is a valid invariant for each procedure call; in order to keep the adversary synchronized in the two programs, we further require that procedure calls return equal values in the two programs, whenever they are called with equal values.

6.4 Union bound reasoning about adversarial programs

The Union Bound Logic from Chapter 5 extends to adversarial programs. A judgment $\models_\beta c : \phi \Rightarrow \psi$ is valid iff $\models_\beta c' : \phi \Rightarrow \psi$ for every concrete refinement c' of c. It can be shown that all the rules of pHL retain soundness under this interpretation.

Moreover, one can define a rule for adversaries. The rule for adversaries is parametrized by a resource policy. Informally, suppose that every call to an oracle $\mathcal{F}$ preserves the invariant θ with probability at least $1 - \beta_{\mathcal{F}}$; then a call to an adversary will preserve the invariant the invariant θ with probability at least $1 - \sum_{\mathcal{F} \in \mathbf{Proc}} q_{\mathcal{F}} \beta_F$, where $q_{\mathcal{F}}$ is the maximal number of queries that $\mathcal{A}$ can make to the procedure $\mathcal{F}$. Formally, the rule for adversaries is:

$$[\textsc{Adv}]\ \frac{\forall \mathcal{F}, y, z.\ \models_{\beta_{\mathcal{F}}} y \leftarrow \mathcal{F}(z) : \theta \Rightarrow \theta}{\models_{\sum_{\mathcal{F} \in \mathbf{Proc}} q_{\mathcal{F}} \beta_F} x \leftarrow \mathcal{A}(e) : \theta \Rightarrow \theta}$$

The soundness of the rule is itself an immediate consequence of the Union Bound principle.

6.5 Computational complexity and termination behavior

Our formalization does not impose any explicit requirement on the computational complexity or termination behavior of adversaries.

7

Tools

This chapter reviews tools based on probabilistic relational Hoare Logic and its variants. We first review tools that implement pRHL, then discuss tools that generate automatically valid proofs in pRHL.

7.1 CertiCrypt

CertiCrypt (Barthe *et al.*, 2009) is a foundational framework built on top of the Coq proof assistant (**Coq**). CertiCrypt consists of two main layers.

The lower layer consists of a deep embedding of a variant of the pWhile language (with recursive procedure calls). CertiCrypt provides two semantics of programs: an operational semantics and a denotational semantics, which are proved equivalent. CertiCrypt also provides instrumented (operational and denotational) semantics, using a cost monad for tracking the execution cost of programs. The instrumented semantics are used for defining the class probabilistic polynomial-time programs. All semantics use the Alea library for modelling sub-distributions Audebaud and Paulin-Mohring (2009);

The upper layer comprises a rich set of tools for manipulating or

reasoning about probabilistic programs. All the tools are implemented in Coq, and are proven correct with respect to the program semantics. The main tools are:

- a variant of probabilistic Relational Hoare Logic with shallow assertions, i.e. assertions are Coq relations over memories. CertiCrypt also provides a relational weakest pre-condition calculus for loop-free programs;

- automated tactics for syntatic failure events. The tactics apply the Failure Event Lemma on two programs G_1 and G_2 that are syntactically equal up to a syntactic failure event, modelled by a boolean flag bad. The tactics checks that the code of the two games differs only after program points setting the flag bad to true and that bad is never reset to false afterwards. If this is the case, it outputs a proof of

$$|\mathbb{P}_{\mathsf{G}_1}[A] - \mathbb{P}_{\mathsf{G}_2}[A]| \leq \max(\mathbb{P}_{\mathsf{G}_1}[F], \mathbb{P}_{G_2}[F])$$

  where $F \triangleq \mathsf{bad} = \top$;

- proof principles and program transformations based on dependence analysis. CertiCrypt implements functions that perform dependence analysis and compute for each statement c and set I of initial variables an over-approximation of the set O of output variables influenced by I, and dually for each statement c and set O of ouput variables an over-approximation of the set I of input variables on which the variables in O depend. These functions are used for automating proofs of observational equivalence of structurally similar programs, and for simplifying proof goals in relational Hoare logic. In addition, the dependence analysis provides an (incomplete) method for justifying code motion.

- program transformations based on dataflow analysis. CertiCrypt implements a generic dataflow analysis over an abstract domain D, modelled as a semi-lattice. Given transfer functions for assignment and branching instructions, and abstract functions for operators of the expression language, CertiCrypt yields a certified optimization

function that receives a statement c and an abstract pre-condition $\delta \in D$, and returns a statement c' and an abstract post-condition δ', such that c and c' are equivalent on memories for δ holds, and δ' holds after executing c (or c'). Abstract post-conditions are used for stating formally the correctness of the optimizer:

$$\models c \sim c' : \Phi \Rightarrow \Psi$$

where $\Phi \triangleq \lambda m_1, m_2.\ m_1 = m_2 \wedge \mathsf{valid}(\delta, m_1)$, $\Psi \triangleq \lambda m_1, m_2.\ m_1 = m_2 \wedge \mathsf{valid}(\delta', m_1)$ and $\mathsf{valid}(\delta, n)$ states that the memory m satisfies the abstract value δ. The correctness of the optimizer is proved using techniques from certified compilation Bertot *et al.*, 2004; Leroy, 2006.

- generic program transformations, including procedure inlining and loop unrolling, and a domain-specific inter-procedural program transformation for eager and lazy sampling. The latter automates a custom proof system for swapping statements **BartheGZ10**
- a (non-relational) program logic for proving upper bounds on the probability of events **BartheGZ10** The logic shares the goals of probabilistic Hoare logic, but is *ad hoc* and also less expressive.

CertiCrypt has been used to prove the security of several prominent cryptographic constructions, including the Full Domain Hash signature (Zanella-Béguelin *et al.*, 2009), the OAEP padding scheme (Barthe *et al.*, 2011b), the Boneh-Franklin identity-based encryption scheme (Barthe *et al.*, 2011c), zero-knowledge protocols (Barthe *et al.*, 2010), and hash functions into elliptic curves (Barthe *et al.*, 2013c).

7.2 EasyCrypt

EasyCrypt (Barthe *et al.*, 2011a; Barthe *et al.*, 2013b) is a proof assistant that combines interactive and automated proofs through a tactic-based proof engine similar to Coq and a back-end to multiple theorem provers and SMT solvers via the Why3 platform (Bobot *et al.*, 2016). EasyCrypt supports program logics for reasoning about probabilistic programs. The main program logics supported by EasyCrypt are probabilistic Relational

Hoare Logic and (an extension of) probabilistic Hoare Logic. Both logics are embedded into a higher-order logic in which top-level reasoning is carried. In particular, the higher-order logic helps glueing together probabilistic inequalities obtained using program logics, or proving mathematical facts that are used by the program logic.

EasyCrypt also provides several mechanisms and proof techniques that are not supported by CertiCrypt:

- an advanced module system for structuring proofs. The module system significantly broadens the scope of formal reasoning, by providing a useful abstraction mechanism for large proofs, and by supporting hierarchical proofs. In the first case, the module system opens the possibility to perform successive reductions locally and to obtain the global constructed adversary automatically, as often featured in pen-and-paper proofs; in absence of such a mechanism, the constructred adversary would have to be carried explicitly through the whole proof, which is at best undesirable, and even potentially blocking. More fundamentally, the module system opens the possibility to build hierarchical proofs. For instance, one can first prove a general theorem using abstract views of cryptographic primitives, then prove instantiations of these primitives secure down to computational assumptions, and then glue the two results together. This kind of modular reasoning is essential for giving machine-checked security proofs of high-level protocols.

  Module types are used to declare the interface of modules: a set of procedures, and their types. Modules themselves consist of a set of procedures, and a memory (a set of global variables, shared by all procedures of the module). A module satisfies a module type if it implements all the procedures declared in the module type. It is possible that a module implement procedures that are not declared in their module type, but such procedures are not exported outside of the module. Module types can also be used to declare functors, i.e. modules parametrized by other modules. Procedures of a functor can use procedures of the parameter modules as oracles. For applications to cryptography, it is however

essential to control the interactions between the procedures of a functor and the procedures of parameter modules; we use module types for specifying which procedures of the parameter modules can be accessed by the procedures of a functor. By itself, the module system provides a mechanism for structuring the definition of cryptographic experiments. For instance, adversaries, security notions, and computational assumptions are all formalized using model. However, the strength of the module system lies in its logical status; specifically, the higher-order logic of EasyCrypt supports (arbitrary) universal and existential quantification over modules. This is the key to compositional reasoning, and is used critically in the most advanced applications of EasyCrypt;

- a theory mechanisms that can be used for organizing and reusing the axiomatizations of different algebraic and data structures such as cyclic groups, finite fields, matrices, finite maps, lists or arrays. In its simplest form, a theory consists of a collection of type and operator declarations, and a set of axioms. However, theories might also contain modules, allowing the definition of libraries of standard games depending on abstract algebraic and data structures.

  Theories enjoy a cloning mechanism that is useful when formalizing examples that involve multiple objects of the same nature, e.g. cyclic groups in bilinear pairings. Moreover, operators of a theory can be realized, i.e. instantiated by expressions, during cloning. Cloning can also be used as a substitute for polymorphic modules.

- generic libraries of advanced cryptographic techniques, used for hybrid arguments, eager/lazy sampling and random oracles.

EasyCrypt has been used to prove the security of several examples, including the Cramer-Shoup encryption scheme (Barthe *et al.*, 2011a), the Merkle-Damgård iterative hash function design (Backes *et al.*, 2012), and the ZAEP encryption scheme (Barthe *et al.*, 2012), the TLS handshake protocol Bhargavan *et al.*, 2014, a generic authenticated key exchange protocol with instantiations to Nets and NAXOS (Barthe

et al., 2015), masked algorithms (Barthe *et al.*, 2016a), and the Helios voting system **CortierDDSSW17**

Remark 7.1. Early versions of EasyCrypt provided an extraction mechanism for generating an independently verifiable CertiCrypt proof of the validity of pRHL judgments, conditioned by the validity of the formulae discharged by the SMT solvers. This mechanism is no longer supported, since EasyCrypt now offers the support of a full-fledged proof assistant. In addition, early versions of EasyCrypt explored alternative trade-offs between control and automation, and provided several automated components: a verification condition generator for pRHL, together with a rudimentary algorithm for inferring relational loop invariants and adversary specifications; a rudimentary algorithm for synthesizing and proving probability bounds.

7.3 AutoGP

7.4 ZooCrypt

The ZooCrypt framework (Barthe *et al.*, 2013a) provides tools for automatically analyzing and synthesizing padding-based encryption schemes. The class of padding-based encryption schemes consists of public-key encryption schemes built from one-way trapdoor permutations and random oracles. In practice, these primitives are often instantiated with the RSA function.

Even though these building blocks are relatively simple and well understood, it is surprisingly difficult to find constructions that are simple, minimize ciphertext expansion and support tight reductions to the security of the employed one-way function. For example, Bellare and Rogaway (1994) proved security against chosen-ciphertext attacks (IND-CCA) for the OAEP scheme under the one-way assumption. Later on, Shoup (2001) proved that it is impossible to reduce the security of OAEP to one-wayness and the proof must therefore be flawed. To regain confidence in the widely used OAEP scheme, Shoup (2001) and Fujisaki *et al.* (2001) developed new proofs for OAEP under stronger assumptions. Additionally, many schemes have been proposed that

improve on various aspects of OAEP, for example by providing security under the weaker one-wayness assumption.

The goal of the ZooCrypt framework is to demonstrate that fully automated game-based proofs and computer-aided design are feasible in the domain of padding-based encryption schemes. ZooCrypt consists of two components: an analyzer that can decide efficiently whether an instance construction is secure, and a synthesizer that implements a smart generation algorithm for candidate instances. The analyzer combines efficient search procedures to prove the security of an instance using a custom proof system, and attack finding procedures based on symbolic models of cryptography. The custom proof system consists of a small number of high level proof rules that formalize the game hops used in such proofs. Using ZooCrypt, we have built a database that contains more than one million padding-based encryption schemes. To build this database, our tool has not only found many new schemes, it has also rediscovered most schemes from the literature (including proofs).

7.5 Computational Indistinguishibility Framework

7.6 Foundational Cryptography Framework

The Foundational Cryptography Framework (FCF) of Petcher and Morrisett (2015b) is an alternative formalization of pRHL in the Coq proof assistant. In contrast to CertiCrypt, FCF provides a lightweight formalization of the programming language, letting users take advantage of the rich specification language of Coq for writing cryptographic games. In Coq parlance, CertiCrypt supports a deep embedding of the pWhile language, whereas FCF is inspired from shallow embedding; however, the formalization of pWhile in FCF also contains elements of a deep embedding in order to reason about the complexity of programs. The approaches deliver different benefits in terms of expressiveness and automation, and are difficult to compare. FCF has been used to mechanize a proof of security for a searchable symmetric encryption scheme (Petcher and Morrisett, 2015a), and for a proof of security of the HMAC message authenthication code (Beringer *et al.*, 2015).

References

Audebaud, P. and C. Paulin-Mohring. 2009. "Proofs of randomized algorithms in Coq". *Sci. Comput. Program.* 74(8): 568–589. DOI: 10.1016/j.scico.2007.09.002. URL: http://dx.doi.org/10.1016/j.scico.2007.09.002.

Backes, M., G. Barthe, M. Berg, B. Grégoire, C. Kunz, M. Skoruppa, and S. Zanella-Béguelin. 2012. "Verified Security of Merkle-Damgård". In: *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*. Ed. by S. Chong. IEEE Computer Society. 354–368. DOI: 10.1109/CSF.2012.14. URL: http://dx.doi.org/10.1109/CSF.2012.14.

Barthe, G., S. Belaid, F. Dupressoir, P. Fouque, B. Grégoire, P. Strub, and R. Zucchini. 2016a. "Strong Non-Interference and Type-Directed Higher-Order Masking". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. Ed. by E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi. ACM. 116–129. DOI: 10.1145/2976749.2978427. URL: http://doi.acm.org/10.1145/2976749.2978427.

Barthe, G., J. M. Crespo, B. Grégoire, C. Kunz, Y. Lakhnech, B. Schmidt, and S. Zanella-Béguelin. 2013a. "Fully automated analysis of padding-based encryption in the computational model". In: *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*. Ed. by A. Sadeghi, V. D. Gligor, and M. Yung. ACM. 1247–1260. DOI: 10.1145/2508859.2516663. URL: http://doi.acm.org/10.1145/2508859.2516663.

Barthe, G., J. M. Crespo, Y. Lakhnech, and B. Schmidt. 2015. "Mind the Gap: Modular Machine-Checked Proofs of One-Round Key Exchange Protocols". In: *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*. Ed. by E. Oswald and M. Fischlin. Vol. 9057. *Lecture Notes in Computer Science*. Springer. 689–718. DOI: 10.1007/978-3-662-46803-6_23. URL: http://dx.doi.org/10.1007/978-3-662-46803-6_23.

Barthe, G., F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P. Strub. 2013b. "EasyCrypt: A Tutorial". In: *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*. Ed. by A. Aldini, J. Lopez, and F. Martinelli. Vol. 8604. *Lecture Notes in Computer Science*. Springer. 146–166. DOI: 10.1007/978-3-319-10082-1_6. URL: http://dx.doi.org/10.1007/978-3-319-10082-1_6.

Barthe, G., M. Gaboardi, B. Grégoire, J. Hsu, and P. Strub. 2016b. "A Program Logic for Union Bounds". In: *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*. Ed. by I. Chatzigiannakis, M. Mitzenmacher, Y. Rabani, and D. Sangiorgi. Vol. 55. *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. 107:1–107:15. DOI: 10.4230/LIPIcs.ICALP.2016.107. URL: http://dx.doi.org/10.4230/LIPIcs.ICALP.2016.107.

Barthe, G., B. Grégoire, D. Hedin, S. Heraud, and S. Zanella-Béguelin. 2010. "A Machine-Checked Formalization of Sigma-Protocols". In: *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010*. IEEE Computer Society. 246–260. DOI: 10.1109/CSF.2010.24. URL: http://dx.doi.org/10.1109/CSF.2010.24.

Barthe, G., B. Grégoire, S. Heraud, F. Olmedo, and S. Zanella-Béguelin. 2013c. "Verified indifferentiable hashing into elliptic curves". *Journal of Computer Security*. 21(6): 881–917. DOI: 10.3233/JCS-130476. URL: http://dx.doi.org/10.3233/JCS-130476.

Barthe, G., B. Grégoire, S. Heraud, and S. Zanella-Béguelin. 2011a. "Computer-Aided Security Proofs for the Working Cryptographer". In: *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*. Ed. by P. Rogaway. Vol. 6841. *Lecture Notes in Computer Science*. Springer. 71–90. DOI: 10.1007/978-3-642-22792-9_5. URL: http://dx.doi.org/10.1007/978-3-642-22792-9_5.

Barthe, G., B. Grégoire, J. Hsu, and P. Strub. 2017. "Coupling proofs are probabilistic product programs". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. Ed. by G. Castagna and A. D. Gordon. ACM. 161–174. URL: http://dl.acm.org/citation.cfm?id=3009896.

Barthe, G., B. Grégoire, Y. Lakhnech, and S. Zanella-Béguelin. 2011b. "Beyond Provable Security Verifiable IND-CCA Security of OAEP". In: *Topics in Cryptology - CT-RSA 2011 - The Cryptographers' Track at the RSA Conference 2011, San Francisco, CA, USA, February 14-18, 2011. Proceedings*. Ed. by A. Kiayias. Vol. 6558. *Lecture Notes in Computer Science*. Springer. 180–196. DOI: 10.1007/978-3-642-19074-2_13. URL: http://dx.doi.org/10.1007/978-3-642-19074-2_13.

Barthe, G., B. Grégoire, and S. 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*. Ed. by Z. Shao and B. C. Pierce. ACM. 90–101. DOI: 10.1145/1480881.1480894. URL: http://doi.acm.org/10.1145/1480881.1480894.

Barthe, G., F. Olmedo, and S. Zanella-Béguelin. 2011c. "Verifiable Security of Boneh-Franklin Identity-Based Encryption". In: *Provable Security - 5th International Conference, ProvSec 2011, Xi'an, China, October 16-18, 2011. Proceedings*. Ed. by X. Boyen and X. Chen. Vol. 6980. *Lecture Notes in Computer Science*. Springer. 68–83. DOI: 10.1007/978-3-642-24316-5_7. URL: http://dx.doi.org/10.1007/978-3-642-24316-5_7.

Barthe, G., D. Pointcheval, and S. Zanella-Béguelin. 2012. "Verified security of redundancy-free encryption from Rabin and RSA". In: *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*. Ed. by T. Yu, G. Danezis, and V. D. Gligor. ACM. 724–735. DOI: 10.1145/2382196.2382272. URL: http://doi.acm.org/10.1145/2382196.2382272.

Bellare, M. and P. Rogaway. 1994. "Optimal Asymmetric Encryption". In: *Advances in Cryptology - EUROCRYPT '94, Workshop on the Theory and Application of Cryptographic Techniques, Perugia, Italy, May 9-12, 1994, Proceedings*. Ed. by A. D. Santis. Vol. 950. *Lecture Notes in Computer Science*. Springer. 92–111. DOI: 10.1007/BFb0053428. URL: http://dx.doi.org/10.1007/BFb0053428.

Bellare, M. and P. 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*. Ed. by S. Vaudenay. Vol. 4004. *Lecture Notes in Computer Science*. Springer. 409–426. DOI: 10.1007/11761679_25. URL: http://dx.doi.org/10.1007/11761679_25.

Beringer, L., A. Petcher, K. Q. Ye, and A. W. Appel. 2015. "Verified Correctness and Security of OpenSSL HMAC". In: *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*. Ed. by J. Jung and T. Holz. USENIX Association. 207–221. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/beringer.

Bertot, Y., B. Grégoire, and X. Leroy. 2004. "A Structured Approach to Proving Compiler Optimizations Based on Dataflow Analysis". In: *Types for Proofs and Programs, International Workshop, TYPES 2004, Jouy-en-Josas, France, December 15-18, 2004, Revised Selected Papers*. Ed. by J. Filliâtre, C. Paulin-Mohring, and B. Werner. Vol. 3839. *Lecture Notes in Computer Science*. Springer. 66–81. DOI: 10.1007/11617990_5. URL: http://dx.doi.org/10.1007/11617990_5.

Bhargavan, K., C. Fournet, M. Kohlweiss, A. Pironti, P. Strub, and S. Z. Béguelin. 2014. "Proving the TLS Handshake Secure (As It Is)". In: *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II*. Ed. by J. A. Garay and R. Gennaro. Vol. 8617. *Lecture Notes in Computer Science*. Springer. 235–255. DOI: 10.1007/978-3-662-44381-1_14. URL: http://dx.doi.org/10.1007/978-3-662-44381-1_14.

Bobot, F., J.-C. Filliâtre, C. Marché, G. Melquiond, and A. Paskevich1. 2016. URL: http://why3.lri.fr/download/manual-0.87.3.pdf.

Cramer, R. and V. Shoup. 1998. "A Practical Public Key Cryptosystem Provably Secure Against Adaptive Chosen Ciphertext Attack". In: *Advances in Cryptology - CRYPTO '98, 18th Annual International Cryptology Conference, Santa Barbara, California, USA, August 23-27, 1998, Proceedings*. Ed. by H. Krawczyk. Vol. 1462. *Lecture Notes in Computer Science*. Springer. 13–25. DOI: 10.1007/BFb0055717. URL: http://dx.doi.org/10.1007/BFb0055717.

Deng, Y. 2015. *Semantics of Probabilistic Processes: An Operational Approach*. Jointly published with Shanghai Jiao Tong University Press. Springer. ISBN: 9783662451977. URL: http://basics.sjtu.edu.cn/~yuxin/publications/book/book.html.

Fujisaki, E., T. Okamoto, D. Pointcheval, and J. Stern. 2001. "RSA-OAEP Is Secure under the RSA Assumption". In: *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*. Ed. by J. Kilian. Vol. 2139. *Lecture Notes in Computer Science*. Springer. 260–274. DOI: 10.1007/3-540-44647-8_16. URL: http://dx.doi.org/10.1007/3-540-44647-8_16.

Halevi, S. 2005. "A plausible approach to computer-aided cryptographic proofs". *IACR Cryptology ePrint Archive*. 2005: 181. URL: http://eprint.iacr.org/2005/181.

Jones, C. and G. D. Plotkin. 1989. "A Probabilistic Powerdomain of Evaluations". 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. 186–195. DOI: 10.1109/LICS.1989.39173. URL: http://dx.doi.org/10.1109/LICS.1989.39173.

Kozen, D. 1981. "Semantics of Probabilistic Programs". *J. Comput. Syst. Sci.* 22(3): 328–350. DOI: 10.1016/0022-0000(81)90036-2. URL: http://dx.doi.org/10.1016/0022-0000(81)90036-2.

Kozen, D. 1985. "A Probabilistic PDL". *J. Comput. Syst. Sci.* 30(2): 162–178. DOI: 10.1016/0022-0000(85)90012-1. URL: http://dx.doi.org/10.1016/0022-0000(85)90012-1.

Leroy, X. 2006. "Formal certification of a compiler back-end or: programming a compiler with a proof assistant". In: *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*. Ed. by J. G. Morrisett and S. L. P. Jones. ACM. 42–54. DOI: 10.1145/1111037.1111042. URL: http://doi.acm.org/10.1145/1111037.1111042.

McIver, A. and C. Morgan. 2005. *Abstraction, Refinement and Proof for Probabilistic Systems. Monographs in Computer Science*. Springer. ISBN: 978-0-387-40115-7. DOI: 10.1007/b138392. URL: http://dx.doi.org/10.1007/b138392.

Petcher, A. and G. Morrisett. 2015a. "A Mechanized Proof of Security for Searchable Symmetric Encryption". In: *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*. Ed. by C. Fournet, M. W. Hicks, and L. Viganò. IEEE Computer Society. 481–494. DOI: 10.1109/CSF.2015.36. URL: http://dx.doi.org/10.1109/CSF.2015.36.

Petcher, A. and G. Morrisett. 2015b. "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*. Ed. by R. Focardi and A. C. Myers. Vol. 9036. *Lecture Notes in Computer Science*. Springer. 53–72. DOI: 10.1007/978-3-662-46666-7_4. URL: http://dx.doi.org/10.1007/978-3-662-46666-7_4.

Shannon, C. 1949. "Communication Theory of Secrecy Systems". *Bell System Technical Journal*. 28(Oct.): 656–715. URL: http://ieeexplore.ieee.org/document/6769090/.

Shoup, V. 2001. "OAEP Reconsidered". In: *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*. Ed. by J. Kilian. Vol. 2139. *Lecture Notes in Computer Science*. Springer. 239–259. DOI: 10.1007/3-540-44647-8_15. URL: http://dx.doi.org/10.1007/3-540-44647-8_15.

Shoup, V. 2004. "Sequences of games: a tool for taming complexity in security proofs". *IACR Cryptology ePrint Archive*. 2004: 332. URL: http://eprint.iacr.org/2004/332.

Zanella-Béguelin, S., G. Barthe, B. Grégoire, and F. Olmedo. 2009. "Formally Certifying the Security of Digital Signature Schemes". In: *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*. IEEE Computer Society. 237–250. DOI: 10.1109/SP.2009.17. URL: http://dx.doi.org/10.1109/SP.2009.17.

Index