The computational SLR: a logic for reasoning about computational indistinguishability

Yu Zhang

Faculty of Information Technology
Macau University of Science and Technology
Macau SAR China
yu.zhang@gmail.com

Abstract. Computational indistinguishability is a notion in complexity-theoretic cryptography and is used to define many security criteria. However, in traditional cryptography, proving computational indistinguishability is usually informal and becomes error-prone when cryptographic constructions are complex. This paper presents a formal proof system based on an extension of Hofmann's SLR language, which can capture probabilistic polynomial-time computations through typing and is sufficient for expressing cryptographic constructions. We in particular define rules that justify directly the computational indistinguishability between programs and prove that these rules are sound with respect to the set-theoretic semantics, hence the standard definition of security. We also show that it is applicable in cryptography by verifying, in our proof system, Goldreich and Micali's construction of pseudorandom generator, and the equivalence between next-bit unpredictability and pseudorandomness.

1 Introduction

Research on the formal verification of cryptographic protocols in recent years has switched its focus from the Dolev-Yao model to the computational model — a more realistic model where criteria for the underlying cryptography are considered. *Computational indistinguishability* is an important notion in cryptography and the computational model of protocols, which is particularly used to define many security criteria. However, proving computational indistinguishability in traditional cryptography is usually done in a paper-and-pencil, semi-formal way. It is often error-prone and becomes unreliable when the cryptographic constructions are complex. This paper aims at designing a formal system that can help us to verify cryptographic proofs. Our ultra goal will be fully or partially automating the verification.

Noticing that computational indistinguishability can be seen as a special notion of equivalence between programs, we make use of techniques from the theory of programming languages, but this requires in the first place a proper language for expressing cryptographic constructions and adversaries. In particular, we shall consider only "feasible" adversaries, precisely, probabilistic programs that terminate within polynomial time. While such a complexity restriction can be easily formulated using the model of Turing-machines, it is by no mean a good model for formal verification. At this point, our attention is drawn to Hofmann's SLR system [7, 8], a functional programming language that implements Bellantoni and Cook's safe recursion [3]. The very nice property about SLR is the characterization of polynomial-time computations through typing. The probabilistic extension of SLR has been studied by Mitchell et al. [10], where functions of the proper type capture the computations that terminate in polynomial time on a probabilistic Turing machine.

Our system is based on the probabilistic extension of SLR, and we develop an axiomatization system with rules justifying the computational indistinguishability between programs. We prove that these rules are sound with respect to the set-theoretic semantics of the language, hence coincide with the traditional definition of computational indistinguishability. Reasoning about cryptographic constructions in the proof system is then purely syntactic, without explicit analysis on the probability of program output and the complexity bound of adversaries.

The rest of the paper is organized as follows: Section 2 introduces the computational SLR — a probabilistic extension of Hofmann's SLR, together with an adapted definition of computational indistinguishability based on the language. In Section 3 we develop the equational proof system and prove the soundness of its rules. Cryptographic examples using the proof system are given in Section 4. Section 5 summarizes related work and Section 6 concludes the paper.

2 The computational SLR

We start by define a language for expressing cryptographic constructions and adversaries, as well as the computational indistinguishability between programs. Due to the complexity consideration, the language should offer a mechanism to capture the class of probabilistic polynomial-time computations. Bellantoni and Cook have proposed a recursion model other than the model of Turing-machines, which is called *safe recursion* and defines exactly functions that are computable in polynomial-time on a Turing-machine [3]. This is an intrinsic, purely syntactic mechanism: variables are divided into two classes — safe variables and normal variables, and safe variables must be instantiated by values that are computed using *only* safe variables; recursion must take place on normal variables and intermediate recursion results are never sent to safe variables. When higher-order functions are concerned, it is also required that step functions must be linear, i.e., intermediate recursive results can be used only once in each step.

Hofmann later developed a functional language called SLR to implement the safe recursion [7, 8]. In particular, he introduces a type system with modality to distinguish between normal variables and safe variables, and linearity to distinguish between normal functions and linear functions. He proves that well-typed functions of a proper type are exactly polynomial-time computable functions. Hofmann's original SLR system has a polymorphic type system, but it is not necessary in cryptography, so in this section we first introduce a non-polymorphic version of Hofmann's SLR system, then extend it to express cryptographic constructions. We shall adapt the definition of the computational indistinguishability in our language.

2.1 The non-polymorphic SLR for bitstrings

Types are defined by:

$$\tau, \tau', \ldots ::= \mathsf{Bits} \mid \tau \times \tau' \mid \tau \otimes \tau' \mid \Box\tau \to \tau' \mid \tau \to \tau' \mid \tau \multimap \tau'.$$

Bits is the base type for bitstrings, and all other types are from Hofmann's language: $\tau \times \tau'$ are cartesian product types, and $\tau \otimes \tau'$ are tensor product types as in linear λ-calculus. There are three sorts of functions: $\Box\tau \to \tau'$ are modal functions with no restriction on the use of arguments; $\tau \to \tau'$ are non-modal functions where arguments must be safe values; $\tau \multimap \tau'$ are linear functions where arguments can be used only once. We also use the aspects of SLR to represent these function spaces — $\tau \xrightarrow{a} \tau'$ is a function type with aspect a, which is $(\text{modal}, \text{nonlinear})$ (noted as $\mathfrak{m}$) for $\Box\tau \to \tau'$, $(\text{nonmodal}, \text{nonlinear})$ (noted as $\mathfrak{n}$) for $\tau \to \tau'$ and $(\text{nonmodal}, \text{linear})$ (noted as $\mathfrak{l}$) for $\tau \multimap \tau'$. The aspects are ordered by $\mathfrak{m} \leq \mathfrak{n} \leq \mathfrak{l}$.

The type system also inherits the sub-typing from SLR and we write $\tau <: \tau'$ if τ is a sub-type of τ'. The sub-typing rules are listed in Figure 1. Note that the last rule, from which we can have $\mathsf{Bits} \to \tau <: \mathsf{Bits} \multimap \tau$, states that bitstrings can be duplicated without violating linearity.

$$\frac{}{\tau <: \tau} \qquad \frac{\tau <: \tau' \quad \tau' <: \tau"}{\tau <: \tau"} \qquad \frac{\tau <: \tau' \quad \sigma <: \sigma'}{\tau \times \sigma <: \tau' \times \sigma'} \qquad \frac{\tau <: \tau' \quad \sigma <: \sigma'}{\tau \otimes \sigma <: \tau' \otimes \sigma'}$$

$$\frac{\tau' <: \tau \quad \sigma <: \sigma' \quad a' \leq a}{\tau \xrightarrow{a} \sigma <: \tau' \xrightarrow{a'} \sigma'} \qquad \frac{\tau <: \tau'}{\mathsf{Bits} \to \tau <: \mathsf{Bits} \multimap \tau'}$$

Fig. 1. Sub-typing rules for the computational OSLR

Expressions of SLR are defined by the following grammar:

$$\begin{array}{rcll}
e_1, e_2, \ldots & ::= & x & \text{atomic variables} \\
& | & \mathtt{nil} & \text{empty bitstring} \\
& | & \mathtt{B}_0 \mid \mathtt{B}_1 & \text{bits} \\
& | & \mathtt{case}_\tau & \text{case distinction} \\
& | & \mathtt{rec}_\tau & \text{safe recursor} \\
& | & \lambda x.e & \text{abstraction} \\
& | & e_1 e_2 & \text{application} \\
& | & \langle e_1, e_2 \rangle & \text{product} \\
& | & \mathtt{proj}_1 e \mid \mathtt{proj}_2 e & \text{product projection} \\
& | & e_1 \otimes e_2 & \text{tensor product} \\
& | & \mathtt{let}\ x \otimes y = e_1\ \mathtt{in}\ e_2 & \text{tensor projection}
\end{array}$$

$\mathtt{B}_0$ and $\mathtt{B}_1$ are two constants for constructing bitstrings: if u is a bitstring, $\mathtt{B}_0 u$ (or $\mathtt{B}_1 u$) is the new bitstring with a bit 0 (or 1) added at the left end of u. We often use $\mathtt{B}$ to denote the bit constructor when its value is irrelevant. Note that in this language we work on real bitstrings, not the number that they represent. For instance, 0 and 00 are two different objects in our language, so the two constants $\mathtt{B}_0$ and $\mathtt{B}_1$ are different from the two successors $\mathtt{S}_0$ and $\mathtt{S}_1$ in Hofmann's system. $\mathtt{case}_\tau$ is the constant for case distinction: $\mathtt{case}_\tau(n, e, f_0, f_1)$ tests the bitstring n and returns e if n is an empty bitstring, $f_0(n')$ if the first bit of n is 0 and the rest is n', and $f_1(n')$ if the first bit of n is 1. $\mathtt{rec}_\tau$ is the constant for recursion on bitstrings: $\mathtt{rec}_\tau(e, f, n)$ returns e if n is empty, and $f(n', \mathtt{rec}_\tau(e, f, n'))$ otherwise, where n' is the part of the bitstring n with its first bit cut off.

Typing assertions of expressions are of the form $\Gamma \vdash t : \tau$, where Γ is a typing context that assigns types and aspects to variables. A context is typically written as a list of bindings $x_1 :^{a_1} \tau_1, \ldots, x_n :^{a_n} \tau_n$, where $a_1, \ldots a_n$ are aspects of $\{\mathfrak{m}, \mathfrak{n}, \mathfrak{l}\}$. Typing rules are given in Figure 2.

2.2 The computational SLR

The probabilistic extension of SLR is studied by Mitchell et al. by adding a random bit oracle to simulate the oracle tape in probabilistic Turing-machines [10]. However, in their language, there is no explicit distinction between probabilistic and purely deterministic functions, so we adopt a different type system from Moggi's computational λ-calculus [12], where probabilistic computations are captured by monadic types. We call the language *computational SLR* and often abbreviate it as *CSLR*.

Types in CSLR are extended with a unary type constructor:

$$\tau ::= \ldots \mid \mathsf{T}\tau.$$

It comes from Moggi's language: a type $\mathsf{T}\tau$ is called a monadic type (or a computation type), for computations that return (if they terminate correctly) values of type τ. In our case, a computation always

$$\frac{}{\mathit{\Gamma}, x :^{a} \tau \vdash x : \tau}\ \textit{T-VAR} \qquad \frac{\mathit{\Gamma} \vdash e : \tau \quad \tau <: \tau'}{\mathit{\Gamma} \vdash e : \tau'}\ \textit{T-SUB} \qquad \frac{\mathit{\Gamma}, x :^{a} \tau \vdash e : \tau'}{\mathit{\Gamma} \vdash \lambda x \,.\, e : \tau \xrightarrow{a} \tau'}\ \textit{T-ABS}$$

$$\frac{\mathit{\Gamma}, \mathit{\Delta}_1 \vdash e_1 : \tau \xrightarrow{a} \tau' \quad \mathit{\Gamma}, \mathit{\Delta}_2 \vdash e_2 : \tau \quad \mathit{\Gamma} \text{ nonlinear} \quad x :^{a'} \sigma \in \mathit{\Gamma}, \mathit{\Delta}_2 \text{ implies } a' \leq a}{\mathit{\Gamma}, \mathit{\Delta}_1, \mathit{\Delta}_2 \vdash e_1 e_2 : \tau'}\ \textit{T-APP}$$

$$\frac{\mathit{\Gamma} \vdash e_1 : \tau_1 \quad \mathit{\Gamma} \vdash e_2 : \tau_2}{\mathit{\Gamma} \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}\ \textit{T-PAIR} \qquad \frac{\mathit{\Gamma} \vdash e : \tau_1 \times \tau_2 \quad i \in \{1, 2\}}{\mathit{\Gamma} \vdash \mathtt{proj}_i(e) : \tau_i}\ \textit{T-PROJ}$$

$$\frac{\mathit{\Gamma}, \mathit{\Delta}_1 \vdash e_1 : \tau_1 \quad \mathit{\Gamma}, \mathit{\Delta}_2 \vdash e_2 : \tau_2 \quad \mathit{\Gamma} \text{ nonlinear}}{\mathit{\Gamma}, \mathit{\Delta}_1, \mathit{\Delta}_2 \vdash e_1 \otimes e_2 : \tau_1 \otimes \tau_2}\ \textit{T-TENSOR}$$

$$\frac{\mathit{\Gamma}, \mathit{\Delta}_1, x :^{\upharpoonleft} \tau_1, y :^{\upharpoonleft} \tau_2 \vdash e : \tau \quad \mathit{\Gamma}, \mathit{\Delta}_2 \vdash e' : \tau_1 \otimes \tau_2 \quad \mathit{\Gamma} \text{ nonlinear}}{\mathit{\Gamma}, \mathit{\Delta}_1, \mathit{\Delta}_2 \vdash \mathtt{let}\ x \otimes y = e'\ \mathtt{in}\ e : \tau}\ \textit{T-LET}$$

$$\frac{}{\mathit{\Gamma} \vdash \mathtt{nil} : \mathsf{Bits}}\ \textit{T-NIL} \qquad \frac{i \in \{1, 2\}}{\mathit{\Gamma} \vdash \mathtt{B}_i : \mathsf{Bits} \multimap \mathsf{Bits}}\ \textit{T-BIT}$$

$$\frac{}{\mathit{\Gamma} \vdash \mathtt{rec}_\tau : \tau \multimap \square(\square\mathsf{Bits} \to \tau \multimap \tau) \to \square\mathsf{Bits} \to \tau}\ \textit{T-REC}$$

$$\frac{}{\mathit{\Gamma} \vdash \mathtt{case}_\tau : \mathsf{Bits} \multimap (\tau \times (\mathsf{Bits} \multimap \tau) \times (\mathsf{Bits} \multimap \tau)) \multimap \tau}\ \textit{T-CASE}$$

Fig. 2. Typing rules for the SLR

terminates and can be probabilistic, hence it will return one of a set of values, each with a certain probability. The sub-typing system is then extended with the rule:

$$\frac{\tau <: \tau'}{\mathsf{T}\tau <: \mathsf{T}\tau'}.$$

Expressions of the computational SLR are extended with three constructions for probabilistic computations:

$$\begin{array}{lll} e_1, e_2, \ldots ::= & \ldots & \text{SLR terms} \\ & \mid \mathtt{rand} & \text{oracle bit} \\ & \mid \mathtt{val}(e) & \text{deterministic computation} \\ & \mid \mathtt{bind}\ x = e_1\ \mathtt{in}\ e_2 & \text{sequential computation} \end{array}$$

The constant $\mathtt{rand}$ returns a random bit 0 or 1, each with the probability $\frac{1}{2}$. $\mathtt{val}(e)$ is the trivial (deterministic) computation which returns e with the probability 1. $\mathtt{bind}\ x = e_1\ \mathtt{in}\ e_2$ is the sequential computation which first computes e_1, binds the value to x, then computes e_2. We sometimes abbreviate the program of the form

$$\mathtt{bind}\ x_1 = e_1\ \mathtt{in}\ \ldots \mathtt{bind}\ x_n = e_n\ \mathtt{in}\ e$$

as

$$\mathtt{bind}\ (\, x_1 = e_1, \ldots, x_n = e_n \,)\ \mathtt{in}\ e.$$

Note that the order of some bindings must be carefully kept in the abbreviated form.

Typing rules for these extra constants and constructions are given in Figure 3. Note that when defining a purely deterministic program in CSLR, it is not sufficient to state that their types does not have monadic components. For instance, the function $\lambda x^{\mathsf{Bits}} \,.\, (\lambda y^{\mathsf{TBits}} \,.\, x)\mathtt{rand}$ has type $\mathsf{Bits} \multimap \mathsf{Bits}$, but it still contains probabilistic computations. Instead, we must show that the program can be defined and typed in (non-probabilistic) SLR, and in that case, we say it is *SLR-definable* and *SLR-typable*.

$$\frac{}{\Gamma \vdash \mathtt{rand} : \mathsf{T}\mathsf{Bits}}\ \textit{T-RAND} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathtt{val}(e) : \mathsf{T}\tau}\ \textit{T-VAL}$$

$$\frac{\begin{array}{c}\Gamma, \Delta_1 \vdash e_1 : \mathsf{T}\tau_1 \quad \Gamma, \Delta_2, x :^{a} \tau_1 \vdash e_2 : \mathsf{T}\tau_2 \\ \Gamma \text{ nonlinear} \quad x :^{a'} \sigma \in \Gamma, \Delta_1 \text{ implies } a' \leq a\end{array}}{\Gamma, \Delta_1, \Delta_2 \vdash \mathtt{bind}\ x = e_1\ \mathtt{in}\ e_2 : \mathsf{T}\tau_2}\ \textit{T-BIND}$$

Fig. 3. Typing rules for the computational SLR

As in some standard typed λ-calculi, we can define a reduction system for the computational SLR, and prove that every closed term has a canonical form. In particular, the canonical form of type Bits is:

$$b ::= \mathtt{nil} \mid \mathtt{B}_0 b \mid \mathtt{B}_1 b.$$

If u is a closed term of type Bits, we write $|u|$ for its length. We define the length of a bitstring on its canonical form b:

$$|\mathtt{nil}| = 0, \qquad |\mathtt{B}_i b| = |b| + 1 \quad (i = 0, 1).$$

2.3 A set-theoretic semantics

We write $\mathbb{B}$ for the set of bitstrings, with a special element ϵ denoting the empty bitstring. When u, v are bitstrings, we write $u \cdot v$ for their concatenation. If A, B are sets, we write $A \times B$ and $A \to B$ for their cartesian product and function space. To interpret the probabilistic computations, we adopt the probabilistic monad defined in [14]: if A is set, we write $\mathcal{D}_A : A \to [0, 1]$ for the set of probability mass functions over A. The original monad in [14] is defined using measures instead of mass functions, and is of type $(2^A \to [0, \infty]) \to [0, \infty]$, where 2^A denotes the set of all subsets of A, so that it can also represent computing probabilities over infinite data structure, not just discrete probabilities. But for the sake of simplicity, in this paper we work on mass functions instead of measures. Note that the monad is not the one defined in [10], which is used to keep track of the bits read from the oracle tape rather than reasoning about probabilities.

When d is a mass function of $\mathcal{D}_A$ and $a \in A$, we also write $\mathbf{Pr}[a \leftarrow d]$ for the probability $d(a)$. If there are finitely many elements in $d \in \mathcal{D}_A$, we can write d as $\{(a_1, p_1), \ldots, (a_n, p_n)\}$, where $a_i \in A$ and $p_i = d(a_i)$.

The detailed definition of the set-theoretic semantics is given in Figure 4.

The very nice property of SLR is the characterization of polynomial-time computations (the class PTIME) through typing:

Theorem 1 (Hofmann [8]). *The set-theoretic interpretations of closed terms of type* $\Box\mathsf{Bits} \to \mathsf{Bits}$ *in SLR define exactly polynomial-time computable functions.*

Mitchell et al. have extended Hofmann's result to the probabilistic version of SLR with a random bit oracle, showing that terms of the same type in their language define exactly the functions that can be computed by a probabilistic Turing machine in polynomial time (the class PPT). Although our language is slightly different from their language OSLR (which does not have computation types), the categorical

Interpretation of types:

$$\begin{aligned}
\llbracket \mathsf{Bits} \rrbracket &= \mathbb{B}\\
\llbracket \tau \times \tau' \rrbracket &= \llbracket \tau \rrbracket \times \llbracket \tau' \rrbracket\\
\llbracket \tau \otimes \tau' \rrbracket &= \llbracket \tau \rrbracket \times \llbracket \tau' \rrbracket\\
\left\llbracket \tau \xrightarrow{a} \tau' \right\rrbracket &= \llbracket \tau \rrbracket \to \llbracket \tau' \rrbracket\\
\llbracket \mathsf{T}\tau \rrbracket &= \mathcal{D}_{\llbracket \tau \rrbracket}
\end{aligned}$$

Interpretation of terms:

$$\begin{aligned}
\llbracket x \rrbracket\rho &= \rho(x)\\
\llbracket \mathtt{nil} \rrbracket\rho &= \epsilon\\
\llbracket \mathtt{B}_i \rrbracket\rho &= \underline{\lambda} v\,.\,(i \cdot v),\ i = 0, 1\\
\llbracket \mathtt{rec}_\tau \rrbracket\rho &= \text{function } f \text{ such that for all } v \in \llbracket \tau \rrbracket,\, u \in \llbracket \mathsf{Bits} \rrbracket,\\
&\quad h \in \llbracket \mathsf{Bits} \rrbracket \to \llbracket \tau \rrbracket \to \llbracket \tau \rrbracket,\\
&\quad f(v, h, \epsilon) = v \text{ and}\\
&\quad f(v, h, i \cdot u) = h(u, f(v, h, u))\\
\llbracket \mathtt{case}_\tau \rrbracket\rho &= \text{function } f \text{ such that for all } v \in \llbracket \tau \rrbracket,\, u \in \llbracket \mathsf{Bits} \rrbracket\\
&\quad h_i \in \llbracket \mathsf{Bits} \rrbracket \to \llbracket \tau \rrbracket\ (i = 0, 1),\\
&\quad f(v, h_0, h_1, \epsilon) = u \text{ and}\\
&\quad f(v, h_0, h_1, i \cdot u) = h_i(u)\\
\llbracket \lambda x\,.\,e \rrbracket\rho &= \underline{\lambda} v\,.\,\llbracket e \rrbracket\rho[x \mapsto v]\\
\llbracket e_1 e_2 \rrbracket\rho &= \llbracket e_1 \rrbracket(\llbracket e_2 \rrbracket\rho)\\
\llbracket \langle e_1, e_2 \rangle \rrbracket\rho\ =\ \llbracket e_1 \otimes e_2 \rrbracket\rho &= (\llbracket e_1 \rrbracket\rho, \llbracket e_2 \rrbracket\rho)\\
\llbracket \mathtt{proj}_i e \rrbracket\rho &= v_i,\ \text{where } \llbracket e \rrbracket\rho = (v_1, v_2)\\
\llbracket \mathtt{let}\ x \otimes y = e_1\ \mathtt{in}\ e_2 \rrbracket\rho &= \llbracket e_2 \rrbracket\rho[x \mapsto v_1, y \mapsto v_2] \text{ where } \llbracket e_1 \rrbracket\rho = (v_1, v_2)\\
\llbracket \mathtt{rand} \rrbracket\rho &= \{(0, \tfrac{1}{2}), (1, \tfrac{1}{2})\}\\
\llbracket \mathtt{val}(e) \rrbracket\rho &= \{(\llbracket e \rrbracket\rho, 1)\}\\
\llbracket \mathtt{bind}\ x = e_1\ \mathtt{in}\ e_2 \rrbracket\rho &= \underline{\lambda} v\,.\,\textstyle\sum_{v' \in \llbracket \tau \rrbracket} \llbracket e_2 \rrbracket\rho[x \mapsto v'](v) \times \llbracket e_1 \rrbracket\rho(v')\\
&\quad \text{where } \tau \text{ is the type of the variable } x \text{ (or } \mathsf{T}\tau \text{ is the type of } e_1\text{).}
\end{aligned}$$

Fig. 4. The set-theoretic semantics for the computational SLR

model that they use to prove the above result can be also used to interpret the computational SLR. In particular, if we follow the traditional encoding of call-by-value λ-calculus into Moggi's computational language, function types $\tau \xrightarrow{a} \tau'$ in OSLR will be encoded as $\tau \xrightarrow{a} \mathsf{T}\tau'$ in CSLR, hence OSLR functions that correspond to PPT computations are actually CSLR functions of type $\square\mathsf{Bits} \to \mathsf{TBits}$. This permits us to reuse the result of [10], adapted for the computational SLR:

Theorem 2 (Mitchell et al. [10]). *The set-theoretic interpretations of closed terms of type* $\square\mathsf{Bits} \to \mathsf{TBits}$ *in CSLR define exactly functions that can be computed by a probabilistic Turing machine in polynomial time.*

2.4 Computational indistinguishability

We say that a closed SLR-term p (of type $\square\mathsf{Bits} \to \mathsf{Bits}$) is *length-sensitive* if for every two bitstrings u_1, u_2 of the same length, i.e. $|u_1| = |u_2|$, it holds that $|p(u_1)| = |p(u_2)|$. When a term p is length-sensitive, we write $|p|$ for the underlying length measure function, i.e., $|p|(n) = |p(u)|$, where $|u| = n$. If p and q are two length-sensitive SLR-functions, we write $|p| < |q|$ for the fact that for all bitstring u, $|p(u)| < |q(u)|$, and similar for $|p| > |q|$, $|p| = |q|$, etc. A length-sensitive function is said *positive* if for every bitstring u, $|p(u)| > |u|$.

We say that a closed CSLR-term p (of type $\square\mathsf{Bits} \to \tau$) is *numerical* if its value depends only on the length of its argument, i.e., $[\![p(u_1)]\!] = [\![p(u_2)]\!]$ if $|u_1| = |u_2|$. Note that we do not introduce the standard numerical functions in the language, so the numerical and length-sensitive SLR-functions will be used to represent the usual polynomials of numerals, and we often abbreviate them as *polynomials*. A numerical polynomial is *canonical* if it returns empty bitstring or all-1 bitstrings only.

Intuitively, two probabilistic functions are computationally indistinguishable, if the probability that any feasible adversary can distinguish them becomes negligible when they take sufficiently large arguments. We adapt the definition of the computational indistinguishability of [6, Definition 3.2.2] in the setting of CSLR.

Definition 1 (Computational indistinguishability). *Two CSLR terms f_1 and f_2, both of type* $\square\mathsf{Bits} \to \mathsf{TBits}$*, are* computationally indistinguishable *(written as $f_1 \simeq f_2$) if for every term $\mathcal{A}$ such that $\vdash \mathcal{A} : \square\mathsf{Bits} \to \mathsf{TBits} \to \mathsf{TBits}$, every positive polynomial p (SLR-typable of* $\square\mathsf{Bits} \to \mathsf{Bits}$*), and all bitstring w such that $|w| \geq n$ (for some $n \in \mathbb{N}$),*

$$|\mathbf{Pr}[\epsilon \leftarrow [\![\mathcal{A}(w, f_1(w))]\!]] - \mathbf{Pr}[\epsilon \leftarrow [\![\mathcal{A}(w, f_2(w))]\!]]| < \frac{1}{|p(w)|}.$$

Note that the second parameter of the adversary must be a computation which can be executed several times. If the adversary were of type $\square\mathsf{Bits} \to \mathsf{Bits} \to \mathsf{TBits}$, it would be too weak since the only way to get the second argument from the programs under testing is $\mathtt{bind}\ x = f_i(w)\ \mathtt{in}\ \mathcal{A}(w, x)$, where the adversary executes the programs only once and uses the value everywhere.

2.5 Examples of PPT functions

Before moving on to develop the logic for reasoning about programs in CSLR, we define some useful PPT functions that will be frequently used in cryptographic constructions.

- The random bitstring generation $\boldsymbol{rs}$:

$$\boldsymbol{rs} \overset{\text{def}}{=} \lambda x : \mathsf{Bits}\,.\,\mathtt{rec}(\mathtt{val}(\mathtt{nil}),\, h_{\boldsymbol{rs}},\, x),$$

where $h_{\boldsymbol{rs}}$ is defined by

$$\begin{aligned} h_{\boldsymbol{rs}} \overset{\text{def}}{=} \lambda m\,.\,\lambda r\,.\,\mathtt{bind}\ (\,b = \mathtt{rand},\ u = r\,)\ \mathtt{in} \\ \mathtt{case}(b, \langle \mathtt{val}(\mathtt{nil}), \lambda x.\mathtt{val}(\mathtt{B}_0 u), \lambda x.\mathtt{val}(\mathtt{B}_1 u)\rangle). \end{aligned}$$

rs receives a bitstring and returns a uniformly random bitstring of the same length. It can be checked that $\vdash h_{\boldsymbol{rs}} : \Box\mathsf{Bits} \to \mathsf{TBits} \multimap \mathsf{TBits}$, hence $\vdash \boldsymbol{rs} : \Box\mathsf{Bits} \to \mathsf{TBits}$. Some of the type checking procedure is given in Figure 2.5.

$$\begin{array}{c}
u :^{\mathfrak{l}} \mathsf{Bits}, x :^{\mathfrak{l}} \mathsf{Bits} \ \vdash\ \mathtt{val}(\mathtt{B}_i u) : \mathsf{TBits} \quad (i = 0, 1) \\
\hline
u :^{\mathfrak{l}} \mathsf{Bits} \ \vdash\ \lambda x.\mathtt{val}(\mathtt{B}_i u) : \mathsf{Bits} \multimap \mathsf{TBits} \\
\hline
u :^{\mathfrak{l}} \mathsf{Bits} \vdash \langle \mathtt{val}(\mathtt{nil}), \lambda x.\mathtt{val}(\mathtt{B}_0 u), \lambda x.\mathtt{val}(\mathtt{B}_1 u)\rangle : \\
\qquad \mathsf{TBits} \times (\mathsf{Bits} \multimap \mathsf{TBits}) \times (\mathsf{Bits} \multimap \mathsf{TBits}) \\
\hline
b :^{\mathfrak{l}} \mathsf{Bits}, u :^{\mathfrak{l}} \mathsf{Bits} \ \vdash\ \mathtt{case}(b, \langle \mathtt{val}(\mathtt{nil}), \lambda x.\mathtt{val}(\mathtt{B}_0 u), \lambda x.\mathtt{val}(\mathtt{B}_1 u)\rangle) : \mathsf{TBits} \\
\hline
m :^{\mathfrak{m}} \mathsf{Bits}, r :^{\mathfrak{l}} \mathsf{TBits} \vdash \mathtt{bind}\ (\,b = \mathtt{rand}, u = r\,)\ \mathtt{in} \\
\qquad \mathtt{case}(b, \langle \mathtt{val}(\mathtt{nil}), \lambda x.\mathtt{val}(\mathtt{B}_0 u), \lambda x.\mathtt{val}(\mathtt{B}_1 u)\rangle) : \mathsf{TBits} \\
\hline
\vdash \lambda m\,.\,\lambda r\,.\,\mathtt{bind}\ (\,b = \mathtt{rand}, u = r\,)\ \mathtt{in} \\
\qquad \mathtt{case}(b, \langle \mathtt{val}(\mathtt{nil}), \lambda x.\mathtt{val}(\mathtt{B}_0 u), \lambda x.\mathtt{val}(\mathtt{B}_1 u)\rangle) : \Box\mathsf{Bits} \to \mathsf{TBits} \multimap \mathsf{TBits}
\end{array}$$

Fig. 5. Type checking of the function $h_{\boldsymbol{rs}}$

If e is a closed program of type TBits and all possible results of e are of the same length, we write $|e|$ for the length of its result bitstrings. Clearly, for any bitstring u, the result bitstrings of $\boldsymbol{rs}(u)$ are of the same length and it can be easily checked that $|\boldsymbol{rs}(u)| = |u|$.

- The string concatenation ***conc***:

$$\boldsymbol{conc} \overset{\text{def}}{=} \lambda x\,.\,\lambda y\,.\,\mathtt{rec}(y, h_{\boldsymbol{conc}}, x),$$

where $h_{\boldsymbol{conc}}$ is defined by

$$h_{\boldsymbol{conc}} \overset{\text{def}}{=} \lambda m\,.\,\lambda r\,.\,\mathtt{case}(m, \langle r, \lambda x.\mathtt{B}_0 r, \lambda x.\mathtt{B}_1 r\rangle).$$

$h_{\boldsymbol{conc}}$ is a purely deterministic, well-typed SLR-function of type $\Box\mathsf{Bits} \to \mathsf{TBits} \multimap \mathsf{TBits}$, hence $\vdash \boldsymbol{conc} : \Box\mathsf{Bits} \to \mathsf{Bits} \multimap \mathsf{Bits}$. Note that ***conc*** can also be defined as a SLR-term of type $\mathsf{Bits} \multimap \Box\mathsf{Bits} \to \mathsf{Bits}$, i.e., it recurs on only one of its argument but it does not matter which one, so we do not distinguish the two forms but only require that one of the two arguments of ***conc*** must be normal (modal). We often abbreviate $\boldsymbol{conc}(u, v)$ as $u \bullet v$.

- Head function ***hd***:

$$\boldsymbol{hd} \overset{\text{def}}{=} \lambda x\,.\,\mathtt{case}(x, \langle \mathtt{nil}, \lambda y.0, \lambda y.1\rangle)$$

Tail function ***tl***:

$$\boldsymbol{tl} \overset{\text{def}}{=} \lambda x\,.\,\mathtt{case}(x, \langle \mathtt{nil}, \lambda y.y, \lambda y.y\rangle)$$

Both ***hd*** and ***tl*** are SLR-definable and SLR-typable of type $\mathsf{Bits} \multimap \mathsf{Bits}$.

- Split function ***split***:

$$\textbf{\textit{split}} \overset{\text{def}}{=} \lambda x \,.\, \lambda n \,.\, \texttt{rec}(\texttt{nil} \otimes x, h_{\textbf{\textit{split}}}, n),$$

  where

$$h_{\textbf{\textit{split}}} \overset{\text{def}}{=} \lambda m \,.\, \lambda r \,.\, \texttt{let}\ v_1 \otimes v_2 = r\ \texttt{in}\ \texttt{case}(v_2, \langle v_1 \otimes v_2, \lambda y.(v_1 \bullet 0) \otimes y, \lambda y.(v_1 \bullet 1) \otimes y \rangle).$$

  split(x, n) splits the bitstring x into two bitstrings, among which the first one is of the length $|n|$ if $|n| \leq |x|$ or x otherwise. It can be checked that ***split*** is SLR-definable and SLR-typable of type $\mathsf{Bits} \multimap \Box\mathsf{Bits} \to \mathsf{Bits} \otimes \mathsf{Bits}$. With ***split*** we can define the prefix and suffix functions:

$$\textbf{\textit{pref}} \overset{\text{def}}{=} \lambda x \,.\, \lambda n \,.\, \texttt{let}\ u_1 \otimes u_2 = \textbf{\textit{split}}(x, n)\ \texttt{in}\ u_1,$$
$$\textbf{\textit{suff}} \overset{\text{def}}{=} \lambda x \,.\, \lambda n \,.\, \texttt{let}\ u_1 \otimes u_2 = \textbf{\textit{split}}(x, n)\ \texttt{in}\ u_2.$$

  Both of the two functions are SLR-definable of type $\mathsf{Bits} \multimap \Box\mathsf{Bits} \to \mathsf{Bits}$.
- Cut function ***cut***:

$$\textbf{\textit{cut}} \overset{\text{def}}{=} \lambda x \,.\, \lambda n \,.\, \textbf{\textit{pref}}(x, \textbf{\textit{suff}}(x, n)).$$

  cut(x, n) cuts the right part of length $|n|$ of the bitstring x. We shall often abbreviate it as $x - n$. ***cut*** is SLR-definable of type $\mathsf{Bits} \multimap \Box\mathsf{Bits} \to \mathsf{Bits}$.

3 The proof system

We present in this section an equational proof system $\mathcal{C}$ on top of CSLR, through which one can justify the computational indistinguishability between CSLR programs at the syntactic level.

The system $\mathcal{C}$ has two sets of rules: the first set (Figure 6) are rules for justifying semantic equivalence between CSLR programs (we write $e_1 \equiv e_2$ if e_1, e_2 are semantic equivalent), and the second set (Figure 7) are rules for justifying computational indistinguishability.

The first set are standard rules in typed λ-calculi, with axioms for probabilistic computations. Rules in the second set are similar as in the logic of Impagliazzo and Kapron [9] (which we shall refer to as the IK-logic in the sequel), where they also define an equational proof system for the computational indistinguishability based on their own arithmetic model. But here we do not have the *EDIT* rule for managing bitstrings, as appears internally in their logic, because in our language, there is no primitive operations for editing bitstrings except the two bit constructor $\mathtt{B}_0, \mathtt{B}_1$. Many bitstring operations are defined as CSLR functions and we have introduced a series of lemmas for bitstring operations (see Section 3.2).

The *H-IND* rule comes from the frequently used hybrid technique in cryptography: if two complex programs can be transformed into a "small" (polynomial) number of hybrids (relatively simpler programs), where the extreme hybrids are exactly the original programs, then proving the computational indistinguishability of the two original programs can be reduced to proving the computational indistinguishability between neighboring hybrids. The *H-IND* in our system is slightly different from that in the IK-logic since we do not have the general primitive that returns uniformly a number which is smaller than a polynomial, but the underlying support from the hybrid technique remains there.

3.1 Soundness of the system $\mathcal{C}$

To show that the system $\mathcal{C}$ is *sound* with respect to the set-theoretic semantics of CSLR, we prove the soundness of the two sets of rules.

Axioms:

$$\frac{}{e \equiv e}\ \textit{AX-REFL} \qquad \frac{}{\texttt{rec}(e_1, e_2, \texttt{nil}) \equiv e_1}\ \textit{AX-REC-NIL}$$

$$\frac{}{\texttt{rec}(e_1, e_2, \mathtt{B}e) \equiv e_2(e, \texttt{rec}(e_1, e_2, e))}\ \textit{AX-REC}$$

$$\frac{}{\texttt{case}(\texttt{nil}, \langle e', e_0, e_1\rangle) \equiv e'}\ \textit{AX-CASE-NIL} \qquad \frac{i = 0, 1}{\texttt{case}(\mathtt{B}_i e, \langle e', e_0, e_1\rangle) \equiv e_i\ e}\ \textit{AX-CASE-i}$$

$$\frac{}{(\lambda x.e)e' \equiv e[e'/x]}\ \textit{AX-}\beta \qquad \frac{x \notin FV(e)}{\lambda x.ex \equiv e}\ \textit{AX-}\eta \qquad \frac{i = 1, 2}{\texttt{proj}_i\langle e_1, e_2\rangle \equiv e_i}\ \textit{AX-PROJ-i}$$

$$\frac{}{\langle \texttt{proj}_1 e, \texttt{proj}_2 e\rangle \equiv e}\ \textit{AX-PAIR} \qquad \frac{}{\texttt{let}\ x_1 \otimes x_2 = e_1 \otimes e_2\ \texttt{in}\ e \equiv e[e_1/x_1, e_2/x_2]}\ \textit{AX-LET}$$

$$\frac{}{(\texttt{let}\ x_1 \otimes x_2 = e\ \texttt{in}\ x_1) \otimes (\texttt{let}\ x_1 \otimes x_2 = e\ \texttt{in}\ x_2) \equiv e}\ \textit{AX-TENSOR}$$

$$\frac{}{\texttt{bind}\ b = \texttt{rand}\ \texttt{in}\ e \equiv \texttt{bind}\ b = \texttt{rand}\ \texttt{in}\ \texttt{case}(b, \langle e', \lambda x.e[0/b], \lambda x.e[1/b]\rangle)}\ \textit{AX-RAND}$$

$$\frac{}{\texttt{bind}\ x = \texttt{val}(e_1)\ \texttt{in}\ e_2 \equiv e_2[e_1/x]}\ \textit{AX-BIND-1} \qquad \frac{}{\texttt{bind}\ x = e\ \texttt{in}\ \texttt{val}(x) \equiv e}\ \textit{AX-BIND-2}$$

$$\frac{}{\texttt{bind}\ x = (\texttt{bind}\ y = e_1\ \texttt{in}\ e_2)\ \texttt{in}\ e_3 \equiv \texttt{bind}\ y = e_1\ \texttt{in}\ \texttt{bind}\ x = e_2\ \texttt{in}\ e_3}\ \textit{AX-BIND-3}$$

Inference rules:

$$\frac{e \equiv e'}{e' \equiv e}\ \textit{SYM} \qquad \frac{e \equiv e' \quad e' \equiv e''}{e \equiv e''}\ \textit{TRANS} \qquad \frac{e_i \equiv e_i' \quad (i = 1, 2, 3)}{\texttt{rec}(e_1, e_2, e_3) \equiv \texttt{rec}(e_1', e_2', e_3')}\ \textit{REC}$$

$$\frac{e_i \equiv e_i' \quad (i = 1, 2, 3, 4)}{\texttt{case}(e_1, \langle e_2, e_3, e_4\rangle) \equiv \texttt{case}(e_1', \langle e_2', e_3', e_4'\rangle)}\ \textit{CASE} \qquad \frac{e \equiv e'}{\lambda x.e \equiv \lambda xe'}\ \textit{ABS}$$

$$\frac{e_1 \equiv e_1' \quad e_2 \equiv e_2'}{e_1 e_2 \equiv e_1' e_2'}\ \textit{APP} \qquad \frac{e \equiv e' \quad i = 1, 2}{\texttt{proj}_i e \equiv \texttt{proj}_i e'}\ \textit{PROJ-i} \qquad \frac{e_1 \equiv e_1' \quad e_2 \equiv e_2'}{\langle e_1, e_2\rangle \equiv \langle e_1', e_2'\rangle}\ \textit{PAIR}$$

$$\frac{e_1 \equiv e_1' \quad e_2 \equiv e_2'}{e_1 \otimes e_2 \equiv e_1' \otimes e_2'}\ \textit{TENSOR} \qquad \frac{e_1 \equiv e_1' \quad e_2 \equiv e_2'}{\texttt{let}\ x \otimes y = e_1\ \texttt{in}\ e_2 \equiv \texttt{let}\ x \otimes y = e_1'\ \texttt{in}\ e_2'}\ \textit{LET}$$

$$\frac{e \equiv e'}{\texttt{val}(e) \equiv \texttt{val}(e')}\ \textit{VAL} \qquad \frac{e_1 \equiv e_1' \quad e_2 \equiv e_2'}{\texttt{bind}\ x = e_1\ \texttt{in}\ e_2 \equiv \texttt{bind}\ x = e_1'\ \texttt{in}\ e_2'}\ \textit{BIND}$$

Fig. 6. System $\mathcal{C}$ rules for program equivalence

$$\frac{\vdash e_1 : \Box\mathsf{Bits} \to \mathsf{TBits} \quad \vdash e_2 : \Box\mathsf{Bits} \to \mathsf{TBits} \quad e_1 \equiv e_2}{e_1 \simeq e_2}\ \textit{EQUIV}$$

$$\frac{e_1 \simeq e_2 \quad e_2 \simeq e_3}{e_1 \simeq e_3}\ \textit{TRANS-INDIST}$$

$$\frac{x :^{\mathsf{n}} \mathsf{Bits}, y :^{\mathsf{n}} \mathsf{Bits} \vdash e : \mathsf{TBits} \quad e_1 \simeq e_2}{\lambda x \,.\, \mathtt{bind}\, y = e_1(x)\ \mathtt{in}\ e \simeq \lambda x \,.\, \mathtt{bind}\, y = e_2(x)\ \mathtt{in}\ e}\ \textit{SUB}$$

$$\frac{\begin{array}{c} x :^{\mathsf{n}} \mathsf{Bits}, n :^{\mathsf{n}} \mathsf{Bits} \vdash e : \mathsf{TBits} \quad \lambda n.e[u/x] \text{ is numerical for all bitstring } u \\ \lambda x \,.\, e[i(x)/n] \simeq \lambda x \,.\, e[\mathtt{B}_1 i(x)/n] \text{ for all canonical polynomial } i \text{ such that } |i| < |p| \end{array}}{\lambda x \,.\, e[\mathtt{nil}/n] \simeq \lambda x \,.\, e[p(x)/n]}\ \textit{H-IND}$$

Fig. 7. System $\mathcal{C}$ rules for computational indistinguishability

Theorem 3 (Soundness of program equivalence rules). *If $\Gamma \vdash e_1 : \tau$, $\Gamma \vdash e_2 : \tau$, and $e_1 \equiv e_2$ is provable in system $\mathcal{C}$, then $[\![e_1]\!]\rho = [\![e_2]\!]\rho$, where $\rho \in [\![\Gamma]\!]$.*

Proof. Most rules for semantic equivalence are standard in typed λ-calculus. The probabilistic monad certifies the axioms for computations. □

Theorem 4 (Soundness of computational indistinguishablity rules). *If $\Gamma \vdash e_1 : \Box\mathsf{Bits} \to \mathsf{TBits}$, $\Gamma \vdash e_2 : \Box\mathsf{Bits} \to \mathsf{TBits}$, and $e_1 \simeq e_2$ is provable in the system $\mathcal{C}$, then e_1 and e_2 are computationally indistinguishable.*

Proof. We prove that rules in Figure 7 are sound. The soundness of the rule *EQUIV* is obvious.

For the rule *TRANS-INDIST*, let $\mathcal{A}$ be an arbitrary (well-typed hence computable in polynomial time) adversary and q be an arbitrary positive polynomial, then we can easily define another polynomial q' such that for all bitstring u, $|q'(u)| = 2|q(u)|$ (e.g., $q' \stackrel{\text{def}}{=} \lambda x \,.\, q(x) \bullet q(x)$, and clearly it is well typed). Because $e_1 \simeq e_2$, according Definition 1, there exists some $n \in \mathbb{N}$ and for any bitstring w such that $|w| \geq n$,

$$|\mathbf{Pr}[\epsilon \leftarrow [\![\mathcal{A}(w, e_1(w))]\!]] - \mathbf{Pr}[\epsilon \leftarrow [\![\mathcal{A}(w, e_2(w))]\!]]| < \frac{1}{|q'(w)|}.$$

Also because $e_2 \simeq e_3$, there exists another $n \in \mathbb{N}$ and for any bitstring w such that $|w| \geq n'$,

$$|\mathbf{Pr}[\epsilon \leftarrow [\![\mathcal{A}(w, e_2(w))]\!]] - \mathbf{Pr}[\epsilon \leftarrow [\![\mathcal{A}(w, e_3(w))]\!]]| < \frac{1}{|q'(w)|}.$$

Without losing generality, we suppose that $n \geq n'$, then for every bitstring w such that $|w| \geq n$,

$$\begin{aligned} &|\mathbf{Pr}[\epsilon \leftarrow [\![\mathcal{A}(w, e_1(w))]\!]] - \mathbf{Pr}[\epsilon \leftarrow [\![\mathcal{A}(w, e_3(w))]\!]]| \\ \leq\ &|\mathbf{Pr}[\epsilon \leftarrow [\![\mathcal{A}(w, e_1(w))]\!]] - \mathbf{Pr}[\epsilon \leftarrow [\![\mathcal{A}(w, e_2(w))]\!]]| \\ &+ |\mathbf{Pr}[\epsilon \leftarrow [\![\mathcal{A}(w, e_2(w))]\!]] - \mathbf{Pr}[\epsilon \leftarrow [\![\mathcal{A}(w, e_3(w))]\!]]| \\ <\ &\frac{1}{|q'(w)|} + \frac{1}{|q'(w)|} \ = \ \frac{1}{|q(w)|}. \end{aligned}$$

Since p is arbitrary, according to Definition 1, $e_1 \simeq e_3$.

To prove the soundness of the rules *SUB*, we assume that there exists an adversary which can computationally distinguish the two terms in the conclusion part, and show that one can also build another adversary which computationally distinguishes the two terms in the premise part. More precisely, for some polynomial p and any integer n, there exists some bitstring w such that $|w| \geq n$ and

$$|\mathbf{Pr}[\epsilon \leftarrow [\![\mathcal{A}(w, f_1(w))]\!]] - \mathbf{Pr}[\epsilon \leftarrow [\![\mathcal{A}(w, f_2(w))]\!]]| \geq \frac{1}{|p(w)|},$$

where f_1 and f_2 are the two programs in the conclusion part of the rule *SUB*. We then build another adversary $\mathcal{A}'$:

$$\mathcal{A}' \stackrel{\text{def}}{=} \lambda z \,.\, \lambda z' \,.\, \mathcal{A}(z, \texttt{bind}\; y = z' \;\texttt{in}\; e),$$

where f is not free in $\mathcal{A}$ and e. According to the set-theoretic semantics,

$$[\![\mathcal{A}'(w, e_i(w))]\!] = [\![\mathcal{A}(w, \texttt{bind}\; y = e_i(w) \;\texttt{in}\; e)]\!],$$

hence

$$\left|\mathbf{Pr}[\epsilon \leftarrow [\![\mathcal{A}'(w, e_1(w))]\!]] - \mathbf{Pr}[\epsilon \leftarrow [\![\mathcal{A}'(w, e_2(w))]\!]]\right| \geq \frac{1}{|p(w)|},$$

which is a contradiction of the premise $e_1 \simeq e_2$.

The soundness of the rule *H-IND* can be proved in a similar way as the proof of TRANS-INDIST. Let $\mathcal{A}$ be an arbitrary well-typed adversary and q be an arbitrary positive polynomial. Define another polynomial: $q' \stackrel{\text{def}}{=} \lambda x \,.\, \texttt{rec}(\texttt{nil}, \lambda m.\lambda r.q'(x) \bullet r, p(x))$. Clearly, for all bitstrings u, $|q'(u)| = |q(u)| \cdot |p(u)|$. Because $\lambda x.e[i(x)/n] \simeq \lambda x.e[\texttt{B}i(x)/n]$ for all canonical numeral i such that $|i| < |p|$, we can find a sufficiently large number $m \in \mathbb{N}$ such that for all bitstring w whose length is larger than m,

$$\begin{aligned}
&|\mathbf{Pr}[\epsilon \leftarrow [\![\mathcal{A}(w, e[\texttt{nil}/n])]\!]] - \mathbf{Pr}[\epsilon \leftarrow [\![\mathcal{A}(w, e[1/n])]\!]]| < \frac{1}{|q'(w)|} \\
&\quad \cdots\cdots \\
&|\mathbf{Pr}[\epsilon \leftarrow [\![\mathcal{A}(w, e[p(w) - 1/n])]\!]] - \mathbf{Pr}[\epsilon \leftarrow [\![\mathcal{A}(w, e[p(w)/n])]\!]]| < \frac{1}{|q'(w)|}.
\end{aligned}$$

Therefore,

$$\begin{aligned}
&|\mathbf{Pr}[\epsilon \leftarrow [\![\mathcal{A}(w, e[\texttt{nil}/n])]\!]] - \mathbf{Pr}[\epsilon \leftarrow [\![\mathcal{A}(w, e[p(w)/n])]\!]]| \\
\leq\; &|\mathbf{Pr}[\epsilon \leftarrow [\![\mathcal{A}(w, e[\texttt{nil}/n])]\!]] - \mathbf{Pr}[\epsilon \leftarrow [\![\mathcal{A}(w, e[1/n])]\!]]| \\
&+ \cdots\cdots \\
&+ |\mathbf{Pr}[\epsilon \leftarrow [\![\mathcal{A}(w, e[p(w) - 1/n])]\!]] - \mathbf{Pr}[\epsilon \leftarrow [\![\mathcal{A}(w, e[p(w)/n])]\!]]| \\
<\; &\frac{1}{|q'(w)|} + \ldots + \frac{1}{|q'(w)|} = \frac{|p(w)|}{|q'(w)|} = \frac{1}{|q(w)|},
\end{aligned}$$

and according to Definition 1, $\lambda x.e[\texttt{nil}/n] \simeq \lambda x.e[p(x)/n]$, since $q, \mathcal{A}$ are arbitrary. □

3.2 Useful lemmas for proving cryptographic constructions

We introduce in this section some useful lemmas that will be frequently used in reasoning about cryptographic constructions. Most of the lemmas are about the indistinguishable programs using random bitstring generation. Note that these are not internal rules of the rpoof system, but we shall name and use them as we do with system $\boldsymbol{\mathcal{C}}$ rules.

Lemma 1. *For every bitstring u, the functions $\lambda x.$**split**(u, x), $\lambda x.$**pref**(u, x), $\lambda x.$**suff**(u, x) and $\lambda x.u - x$ are numerical polynomials.*

Proof. We prove only the the function $\boldsymbol{split}(u)$ — proofs for all others are similar.

We need to prove that, for all bitstrings n, m such that $|n| = |m|$, $[\![\boldsymbol{split}(u,n)]\!] = [\![\boldsymbol{split}(u,m)]\!]$, or $\boldsymbol{split}(u,n) \equiv \boldsymbol{split}(u,m)$ according to Theorem 3. The proof is an induction on the length of the argument n. The case where $|n| = 0$ is clear. When $|n| > 0$, suppose that $n \equiv \mathtt{B}n'$ and $m \equiv \mathtt{B}m'$, then

$$\begin{aligned}\boldsymbol{split}(u,\mathtt{B}n') &\equiv \mathtt{let}\ v_1 \otimes v_2 = \boldsymbol{split}(u,n')\ \mathtt{in}\ \mathtt{case}(v_2, \langle v_1 \otimes v_2, \lambda y.(v_1{\bullet}0) \otimes y, \lambda y.(v_1{\bullet}1) \otimes y\rangle)\\ &\equiv \mathtt{let}\ v_1 \otimes v_2 = \boldsymbol{split}(u,m')\ \mathtt{in}\ \mathtt{case}(v_2, \langle v_1 \otimes v_2, \lambda y.(v_1{\bullet}0) \otimes y, \lambda y.(v_1{\bullet}1) \otimes y\rangle)\\ &\equiv \boldsymbol{split}(u,\mathtt{B}m')\end{aligned}$$

□

Lemma 2 (HEAD-TAIL). *For all bitstrings b and u such that $|b| = 1$,*

$$\boldsymbol{hd}(b{\bullet}u) \equiv b, \qquad \boldsymbol{tl}(b{\bullet}u) \equiv u$$

Proof. Both can be easily deduced from their definitions. □

Lemma 3 (SPLIT-1). *For all bitstrings u, u', there exist bitstrings u_1, u_2 such that $\boldsymbol{split}(u,u') \equiv u_1 \otimes u_2$ and $|u_1| + |u_2| = |u|$. If $|u'| \leq |u|$, then $|u_1| = |u'|$.*

Proof. We prove by the induction on u'. Obviously, the lemma holds when $u' = \mathtt{nil}$. Consider the induction step:

$$\begin{aligned}\boldsymbol{split}(u,\mathtt{B}u') &\equiv \mathtt{rec}(\mathtt{nil} \otimes u, h_{\boldsymbol{split}}, \mathtt{B}u')\\ &\equiv \mathtt{let}\ v_1 \otimes v_2 = \boldsymbol{split}(u,u')\ \mathtt{in}\\ &\quad\ \mathtt{case}(v_2, v_1 \otimes v_2, \lambda y.(v_1{\bullet}0) \otimes y, \lambda y.(v_1{\bullet}1) \otimes y)\\ &\equiv \mathtt{case}(u_2, u_1 \otimes u_2, \lambda y.(u_1{\bullet}0) \otimes y, \lambda y.(u_1{\bullet}1) \otimes y)\\ &\qquad \text{(by the induction hypothesis, we suppose } \boldsymbol{split}(u,u') \equiv u_1 \otimes u_2)\end{aligned}$$

By induction hypothesis, $|u_2| = |u| - |u_1| = |u| - |u'|$. If $|u'| = |u|$, then $|u_2| = 0$, i.e. $u_2 \equiv \mathtt{nil}$, and $|u_1| = |u|$, hence

$$\boldsymbol{split}(u,\mathtt{B}u') \equiv \mathtt{case}(\mathtt{nil}, u_1 \otimes \mathtt{nil}, \lambda y.(u_1{\bullet}0) \otimes y, \lambda y.(u_1{\bullet}1) \otimes y) \equiv u_1 \otimes \mathtt{nil},$$

and $|u_1| + |\mathtt{nil}| = |u|$. If $|u'| < |u|$, then $|u_2| = |u| - |u'| > 0$, so there exists a bitstring u_2' such that $u_2 \equiv \mathtt{B}u_2'$, hence

$$\boldsymbol{split}(u,\mathtt{B}u') \equiv \mathtt{case}(\mathtt{B}u_2', u_1 \otimes \mathtt{nil}, \lambda y.(u_1{\bullet}0) \otimes y, \lambda y.(u_1{\bullet}1) \otimes y) \equiv (u_1{\bullet}\mathtt{B}) \otimes u_2',$$

and $|u_1{\bullet}\mathtt{B}| + |u_2'| = |u_1| + 1 + |u_2| - 1 = |u|$. Also $|u_1{\bullet}\mathtt{B}| = |u_1| + 1 = |u'| + 1 = |\mathtt{B}u'|$, since $|\mathtt{B}u'| \leq |u|$. □

Lemma 4 (SPLIT-2). *For all bitstrings u and u' sch that $|u'| \geq |u|$,*

$$\boldsymbol{split}(u,\mathtt{nil}) \equiv \mathtt{nil} \otimes u, \qquad \boldsymbol{split}(u,u') \equiv u \otimes \mathtt{nil}.$$

Proof. Firstly, for every bitstring u,

$$\boldsymbol{split}(u,\mathtt{nil}) \equiv \mathtt{rec}(\mathtt{nil} \otimes u, h_{\boldsymbol{split}}, \mathtt{nil}) \equiv \mathtt{nil} \otimes u.$$

Because

$$\begin{aligned}\boldsymbol{split}(u, \mathtt{B}_0u') &\equiv \mathtt{rec}(\mathtt{nil} \otimes u, h_{\boldsymbol{split}}, \mathtt{B}_0u')\\ &\equiv \mathtt{let}\ v_1 \otimes v_2 = \boldsymbol{split}(u, u')\ \mathtt{in}\\ &\quad\ \mathtt{case}(v_2, v_1 \otimes v_2, \lambda y.(v_1 \bullet 0) \otimes y, \lambda y.(v_2 \bullet 1) \otimes y)\\ &\equiv \mathtt{rec}(\mathtt{nil} \otimes u, h_{\boldsymbol{split}}, \mathtt{B}_1u')\\ &\equiv \boldsymbol{split}(u, \mathtt{B}_1u'),\end{aligned}$$

it holds that for every bitstring u_1, u_2 such that $|u_1| = |u_2|$, $\boldsymbol{split}(u, u_1) \equiv \boldsymbol{split}(u, u_2)$.

For every bitstrings u and u' such that $|u'| = |u|$, $\boldsymbol{split}(u, u') \equiv u_1 \otimes u_2$ and $|u_1| = |u'|$ by Lemma 3, then $|u_2| = |u| - |u_1| = 0$, hence $u_2 \equiv \mathtt{nil}$, i.e., $\boldsymbol{split}(u, u') \equiv u_1 \otimes \mathtt{nil}$. □

Corollary 1 (PREF). *For all bitstrings u and u' such that $|u'| \geq |u|$,*

$$\boldsymbol{pref}(u, \mathtt{nil}) \equiv \mathtt{nil}, \qquad \boldsymbol{pref}(u, u') \equiv u.$$

Proof. For every bitstring u,

$$\begin{aligned}\boldsymbol{pref}(u, \mathtt{nil}) &\overset{\text{def}}{\equiv} \mathtt{let}\ u_1 \otimes u_2 = \boldsymbol{split}(u, \mathtt{nil})\ \mathtt{in}\ u_1\\ &\equiv \mathtt{let}\ u_1 \otimes u_2 = \mathtt{nil} \otimes u\ \mathtt{in}\ u_1\\ &\equiv \mathtt{nil},\end{aligned}$$

and for every bitstring u' such that $|u'| \geq |u|$,

$$\begin{aligned}\boldsymbol{pref}(u, u') &\overset{\text{def}}{\equiv} \mathtt{let}\ u_1 \otimes u_2 = \boldsymbol{split}(u, u')\ \mathtt{in}\ u_1\\ &\equiv \mathtt{let}\ u_1 \otimes u_2 = u \otimes \mathtt{nil}\ \mathtt{in}\ u_1\\ &\equiv u.\end{aligned}$$

□

Corollary 2 (SUFF). *For all bitstrings u and u' such that $|u'| \geq |u|$,*

$$\boldsymbol{suff}(u, \mathtt{nil}) \equiv u, \qquad \boldsymbol{suff}(u, u') \equiv \mathtt{nil}.$$

Proof. Similar as in Corollary 1. □

Lemma 5 (CUT). *For all bitstrings u and u' such that $|u'| \geq |u|$,*

$$u - \mathtt{nil} \equiv u, \qquad u - u' \equiv \mathtt{nil}.$$

Proof. The first assertion:

$$u - \mathtt{nil} \equiv \boldsymbol{pref}(u, \boldsymbol{suff}(u, \mathtt{nil})) \equiv \boldsymbol{pref}(u, u) \equiv u.$$

The second assertion:

$$u - u' \equiv \boldsymbol{pref}(u, \boldsymbol{suff}(u, u')) \equiv \boldsymbol{pref}(u, \mathtt{nil}) \equiv \mathtt{nil}.$$

□

Lemma 6 (RS-EQUIV). *For every bitstrings u and v such that $|u| = |v|$, $\boldsymbol{rs}(u) \equiv \boldsymbol{rs}(v)$.*

Proof. We prove by induction on the length of u, v. When $|u| = |v| = 0$, i.e., $u \equiv v \equiv \mathtt{nil}$, clearly $\boldsymbol{rs}(u) \equiv \boldsymbol{rs}(v)$.

For the induction step, suppose that $u \equiv \mathtt{B}u'$ and $v \equiv \mathtt{B}'v'$, hence $|u'| = |v'|$, and by the induction hypothesis, $\boldsymbol{rs}(u') \equiv \boldsymbol{rs}(v')$. Then,

$$\begin{aligned}
\boldsymbol{rs}(\mathtt{B}u') &\equiv \mathtt{rec}(\mathtt{val}(\mathtt{nil}), h_{\boldsymbol{rs}}, \mathtt{B}u') \\
&\equiv h_{\boldsymbol{rs}}(u', \boldsymbol{rs}(u')) \\
&\equiv \mathtt{bind}\ (\, y = \boldsymbol{rs}(u'), b = \mathtt{rand}\,)\ \mathtt{in} \\
&\quad\ \mathtt{case}(b, \mathtt{val}(\mathtt{nil}), \lambda x.\mathtt{val}(\mathtt{B}_0 y), \lambda x.\mathtt{val}(\mathtt{B}_1 y)) \\
&\equiv \mathtt{bind}\ (\, y = \boldsymbol{rs}(u'), b = \mathtt{rand}\,)\ \mathtt{in}\ \mathtt{val}(b \bullet y),
\end{aligned}$$

and similarly,

$$\begin{aligned}
\boldsymbol{rs}(\mathtt{B}'v') &\equiv \mathtt{bind}\ (\, y = \boldsymbol{rs}(v'), b = \mathtt{rand}\,)\ \mathtt{in}\ \mathtt{val}(b \bullet y) \\
&\equiv \mathtt{bind}\ (\, y = \boldsymbol{rs}(u'), b = \mathtt{rand}\,)\ \mathtt{in}\ \mathtt{val}(b \bullet y) \\
&\equiv \boldsymbol{rs}(\mathtt{B}u'),
\end{aligned}$$

therefore, $\boldsymbol{rs}(u) \equiv \boldsymbol{rs}(v)$ for all bitstrings $|u| = |v|$. □

Lemma 7 (RS-CONCAT). *For all bitstrings u and v,*

$$\mathtt{bind}\ (\, x = \boldsymbol{rs}(u), y = \boldsymbol{rs}(v)\,)\ \mathtt{in}\ \mathtt{val}(x \bullet y) \equiv \boldsymbol{rs}(u \bullet v).$$

Proof. We prove by induction on the length of u. When $|u| = 0$, i.e., $u \equiv \mathtt{nil}$,

$$\begin{aligned}
&\mathtt{bind}\ (\, x = \boldsymbol{rs}(\mathtt{nil}), y = \boldsymbol{rs}(v)\,)\ \mathtt{in}\ \mathtt{val}(x \bullet y) \\
\equiv\ &\mathtt{bind}\ y = \boldsymbol{rs}(v)\ \mathtt{in}\ \mathtt{val}(\mathtt{nil} \bullet y) \ \equiv\ \boldsymbol{rs}(v) \ \equiv\ \boldsymbol{rs}(\mathtt{nil} \bullet v).
\end{aligned}$$

For the induction step, suppose that $u \equiv \mathtt{B}u'$ and by induction

$$\mathtt{bind}\ (\, x = \boldsymbol{rs}(u'), y = \boldsymbol{rs}(v)\,)\ \mathtt{in}\ \mathtt{val}(x \bullet y) \equiv \boldsymbol{rs}(u' \bullet v),$$

then

$$\begin{aligned}
&\mathtt{bind}\ (\, x = \boldsymbol{rs}(\mathtt{B}u'), y = \boldsymbol{rs}(v)\,)\ \mathtt{in}\ \mathtt{val}(x \bullet y) \\
\equiv\ &\mathtt{bind}\ (\, x = \mathtt{bind}\ (\, x' = \boldsymbol{rs}(u'), b = \mathtt{rand}\,)\ \mathtt{in}\ \mathtt{val}(b \bullet x'), y = \boldsymbol{rs}(v)\,)\ \mathtt{in}\ \mathtt{val}(x \bullet y) \\
\equiv\ &\mathtt{bind}\ (\, x' = \boldsymbol{rs}(u'), b = \mathtt{rand}, y = \boldsymbol{rs}(v)\,)\ \mathtt{in}\ \mathtt{val}(b \bullet x' \bullet y) \\
\equiv\ &\mathtt{bind}\ b = \mathtt{rand}\ \mathtt{in}\ \mathtt{bind}\ z = \boldsymbol{rs}(u' \bullet v)\ \mathtt{in}\ \mathtt{val}(b \bullet z) \\
\equiv\ &\boldsymbol{rs}(\mathtt{B}(u' \bullet v)) \\
\equiv\ &\boldsymbol{rs}((\mathtt{B}u') \bullet v) \qquad \text{(because } |\mathtt{B}(u' \bullet v)| = |(\mathtt{B}u') \bullet v|\text{).}
\end{aligned}$$

Lemma 8 (RS-COMMUT). *For all bitstrings u and v,*

$$\mathtt{bind}\ (\, x = \boldsymbol{rs}(u), y = \boldsymbol{rs}(v)\,)\ \mathtt{in}\ \mathtt{val}(x \bullet y) \equiv \mathtt{bind}\ (\, x = \boldsymbol{rs}(u), y = \boldsymbol{rs}(v)\,)\ \mathtt{in}\ \mathtt{val}(y \bullet x)$$

Proof.

$$\begin{aligned}
&\texttt{bind}\ (\,x = \boldsymbol{rs}(u), y = \boldsymbol{rs}(v)\,)\ \texttt{in}\ \texttt{val}(x \bullet y) \\
\equiv\ &\boldsymbol{rs}(u \bullet v) \qquad \text{(by the rule } \textit{RS-CONCAT}\text{)} \\
\equiv\ &\boldsymbol{rs}(v \bullet u) \qquad \text{(by the rule } \textit{RS-EQUIV} \text{ because } |u \bullet v| = |v \bullet u|\text{)} \\
\equiv\ &\texttt{bind}\ (\,x = \boldsymbol{rs}(u), y = \boldsymbol{rs}(v)\,)\ \texttt{in}\ \texttt{val}(y \bullet x) \qquad \text{(by the rule } \textit{RS-CONCAT}\text{)}
\end{aligned}$$

Lemma 9 (RS-HEAD). $\texttt{bind}\ x = \boldsymbol{rs}(\mathtt{B}u)\ \texttt{in}\ \texttt{val}(\boldsymbol{hd}(x)) \equiv \texttt{rand}$.

Proof. First, for every bitstring u:

$$\begin{aligned}
\boldsymbol{rs}(\mathtt{B}u) &\equiv \texttt{rec}(\texttt{val}(\texttt{nil}), h_{\boldsymbol{rs}}, \mathtt{B}u) \\
&\equiv h_{\boldsymbol{rs}}(u, \boldsymbol{rs}(u)) \\
&\equiv \texttt{bind}\ u' = \boldsymbol{rs}(u)\ \texttt{in} \\
&\quad\ \texttt{bind}\ b = \texttt{rand}\ \texttt{in}\ \texttt{case}(b, \texttt{val}(\texttt{nil}), \lambda x.\texttt{val}(\mathtt{B}_0 u'), \lambda x.\texttt{val}(\mathtt{B}_1 u')) \\
&\equiv \texttt{bind}\ u' = \boldsymbol{rs}(u)\ \texttt{in}\ \texttt{bind}\ b = \texttt{rand}\ \texttt{in}\ \texttt{val}(b \bullet u').
\end{aligned}$$

hence,

$$\begin{aligned}
&\texttt{bind}\ x = \boldsymbol{rs}(\mathtt{B}u)\ \texttt{in}\ \texttt{val}(\boldsymbol{hd}(x)) \\
\equiv\ &\texttt{bind}\ (\,x = \texttt{bind}\ (\,u' = \boldsymbol{rs}(u), b = \texttt{rand}\,)\ \texttt{in}\ \texttt{val}(b \bullet u')\,)\ \texttt{in}\ \texttt{val}(\boldsymbol{hd}(x)) \\
\equiv\ &\texttt{bind}\ (\,u' = \boldsymbol{rs}(u), b = \texttt{rand}\,)\ \texttt{in}\ \texttt{val}(\boldsymbol{hd}(b \bullet u')) \\
\equiv\ &\texttt{bind}\ (\,u' = \boldsymbol{rs}(u), b = \texttt{rand}\,)\ \texttt{in}\ \texttt{val}(b) \\
\equiv\ &\texttt{rand}.
\end{aligned}$$

Lemma 10 (RS-TAIL). $\texttt{bind}\ x = \boldsymbol{rs}(\mathtt{B}u)\ \texttt{in}\ \texttt{val}(\boldsymbol{tl}(x)) \equiv \boldsymbol{rs}(u)$.

Proof. Similar to the proof of Lemma 9. □

Lemma 11 (RS-SPLIT). *For all bitstrings u and v such that $|u| \geq |v|$,*

$$\begin{aligned}
\texttt{bind}\ x = \boldsymbol{rs}(u)\ \texttt{in}\ \texttt{val}(\boldsymbol{pref}(x, v)) \equiv \boldsymbol{rs}(\boldsymbol{pref}(u, v)), \\
\texttt{bind}\ x = \boldsymbol{rs}(u)\ \texttt{in}\ \texttt{val}(\boldsymbol{suff}(x, v)) \equiv \boldsymbol{rs}(\boldsymbol{suff}(u, v)).
\end{aligned}$$

Proof. Proof of the first assertion:

$$\begin{aligned}
&\texttt{bind}\ x = \boldsymbol{rs}(u)\ \texttt{in}\ \texttt{val}(\boldsymbol{pref}(x, v)) \\
\equiv\ &\texttt{bind}\ x = \boldsymbol{rs}(\boldsymbol{pref}(u, v) \bullet \boldsymbol{suff}(u, v))\ \texttt{in}\ \texttt{val}(\boldsymbol{pref}(x, v)) \\
&\quad \text{(by the rule } \textit{RS-EQUIV}\text{, since } |u| = |\boldsymbol{pref}(u, v) \bullet \boldsymbol{suff}(u, v)|\text{)} \\
\equiv\ &\texttt{bind}\ (\,x_1 = \boldsymbol{rs}(\boldsymbol{pref}(u, v)), x_2 = \boldsymbol{rs}(\boldsymbol{suff}(u, v))\,)\ \texttt{in}\ \texttt{val}(\boldsymbol{pref}(x_1 \bullet x_2, v)) \\
&\quad \text{(by the rule } \textit{RS-CONCAT}\text{)} \\
\equiv\ &\texttt{bind}\ (\,x_1 = \boldsymbol{rs}(\boldsymbol{pref}(u, v)), x_2 = \boldsymbol{rs}(\boldsymbol{suff}(u, v))\,)\ \texttt{in}\ \texttt{val}(x_1) \\
&\quad (\boldsymbol{pref}(x_1 \bullet x_2, v) \equiv x_1 \text{ as } |x_1| = |\boldsymbol{pref}(u, v)| = |v|) \\
\equiv\ &\boldsymbol{rs}(\boldsymbol{pref}(u, v)).
\end{aligned}$$

Similarly one can prove the second assertion.

Lemma 12 (RS-CUT). *For all bitstrings u and u' such that $|u'| \leq |u|$,*

$$\texttt{bind}\, x = \boldsymbol{rs}(u)\ \texttt{in}\ \texttt{val}(x - u') \equiv \boldsymbol{rs}(u - u').$$

Proof.

$$\begin{array}{l}
\quad\ \texttt{bind}\, x = \boldsymbol{rs}(u)\ \texttt{in}\ \texttt{val}(x - u') \\
\equiv \texttt{bind}\, x = \boldsymbol{rs}(u)\ \texttt{in}\ \texttt{val}(\boldsymbol{pref}(x, \boldsymbol{suff}(x, u'))) \\
\equiv \texttt{bind}\, x = \boldsymbol{rs}(u)\ \texttt{in}\ \texttt{val}(\boldsymbol{pref}(x, \boldsymbol{suff}(u, u'))) \\
\qquad \text{(because } \boldsymbol{pref}(x) \text{ is a numeral polynomial and } |\boldsymbol{suff}(x, u')| = |\boldsymbol{suff}(u, u')| \text{ since } |x| = |u|\text{)} \\
\equiv \boldsymbol{rs}(\boldsymbol{pref}(u, \boldsymbol{suff}(u, u'))) \qquad \text{(by the rule } \textit{RS-SPLIT}\text{)} \\
\equiv \boldsymbol{rs}(u - u')
\end{array}$$

Lemma 13 (RS-NEXT-BIT). *For all bitstrings u and i such that $|i| < |u|$,*

$$\boldsymbol{rs}(\boldsymbol{pref}(u, \mathtt{B}i)) \equiv \boldsymbol{rs}(\mathtt{B}\boldsymbol{pref}(u, i)).$$

Proof. According to Lemma 3, because $|\mathtt{B}i| \leq |u|$,

$$|\boldsymbol{pref}(u, \mathtt{B}i)| = |\mathtt{B}i| = |i| + 1 = |\boldsymbol{pref}(u, i)| + 1 = |\mathtt{B}\boldsymbol{pref}(u, i)|,$$

hence $\boldsymbol{split}(\boldsymbol{pref}(u, \mathtt{B}i)) \equiv \boldsymbol{rs}(\mathtt{B}\boldsymbol{pref}(u, i))$ since $\lambda x.\boldsymbol{split}(u, x)$ is a numerical polynomial. □

4 Cryptographic examples

Several cryptographic examples are presented in this section and their proofs of correctness is reformulated in the proof system that we define in the previous section.

4.1 Pseudorandom generators

Our first example is verifying, in our proof system, Goldreich and Micali's construction of pseudorandom generator [6]. This example also appears in [9], but their proof has a subtle flaw (see Section 5 for explanation).

We first reformulate in CSLR the standard definition of pseudorandom generator [6, Definition 3.3.1].

Definition 2 (Pseudorandom Generator). *A* pseudorandom generator *is a length-sensitive SLR term* $\vdash g : \Box\mathsf{Bits} \to \mathsf{Bits}$ *such that* $|g(s)| > |s|$ *for every bitstring* s *and,*

$$\lambda x\,.\,\texttt{bind}\, u = \boldsymbol{rs}(x)\ \texttt{in}\ \texttt{val}(g(u)) \simeq \lambda x\,.\,\boldsymbol{rs}(g(x)).$$

If g is a pseudorandom generator, we call $|g|$ its *expansion factor*.

We recall the construction of Goldreich and Micali [6] (reformulated in CSLR): Suppose that g_1 is a PRG with the expansion factor $|g_1|(x) = x + 1$, i.e.,

$$\lambda x\,.\,\texttt{bind}\, u = \boldsymbol{rs}(x)\ \texttt{in}\ \texttt{val}(g_1(x)) \simeq \lambda x\,.\,\boldsymbol{rs}(\mathtt{B}x).$$

Let $B(x)$ be the function returning the first bit of $g_1(x)$, and $R(x)$ returning the rest bits:

$$B \overset{\text{def}}{=} \lambda x\,.\,\boldsymbol{hd}(g_1(x)), \qquad\qquad R \overset{\text{def}}{=} \lambda x\,.\,\boldsymbol{tl}(g_1(x)).$$

Clearly, both B and R are well typed functions (of the same type $\Box\mathsf{Bits} \to \mathsf{Bits}$). We then define a SLR-function G:

$$G \overset{\mathrm{def}}{=} \lambda u\,.\,\lambda n\,.\,\mathtt{rec}(\mathtt{nil}, \lambda m\,.\,\lambda r\,.\,r \bullet B(R'(u,m)), n),$$

where the function R' is defined as:

$$R' \overset{\mathrm{def}}{=} \lambda u\,.\,\lambda n\,.\,\mathtt{rec}(u, \lambda m\,.\,\lambda r\,.\,R(\boldsymbol{pref}(r,u)), n).$$

It can also be checked that both G and R' are well typed SLR-terms (of type $\Box\mathsf{Bits} \to \Box\mathsf{Bits} \to \mathsf{Bits}$).

We first prove the following property about the function G:

Lemma 14. *For every bitstring n,*

$$\begin{array}{c}\lambda x\,.\,\mathtt{bind}\; u = \boldsymbol{rs}(x)\;\mathtt{in}\\ \mathtt{val}(G(u,\mathtt{B}n))\end{array} \simeq \begin{array}{c}\lambda x\,.\,\mathtt{bind}\;(\,b = \mathtt{rand}, u = \boldsymbol{rs}(x)\,)\;\mathtt{in}\\ \mathtt{val}(b \bullet G(u,n))\end{array}.$$

Proof. Because $R \overset{\mathrm{def}}{=} \lambda x\,.\,\boldsymbol{tl}(g_1(x))$, we can conclude that for every bitstring u, $|R(u)| = |u|$ since $|g_1(u)| = |u|+1$. We then show that for any bitstrings u and n, $R'(u,n) \equiv R^{|n|}(u)$. This can be done by induction on $|n|$: when $|n| = 0$, i.e., $n = \mathtt{nil}$,

$$R'(u,\mathtt{nil}) \equiv \mathtt{rec}(u, \lambda m\,.\,\lambda r\,.\,R(\boldsymbol{pref}(r,u)), \mathtt{nil}) \equiv u;$$

when $n = \mathtt{B}n'$ for some bitstring n', i.e., $|n| = |n'|+1$,

$$\begin{aligned} R'(u,\mathtt{B}n') &\equiv \mathtt{rec}(u, \lambda m\,.\,\lambda r\,.\,R(\boldsymbol{pref}(r,u)), \mathtt{B}n') \\ &\equiv R(\boldsymbol{pref}(R'(u,n'),u)) \\ &\equiv R(\boldsymbol{pref}(R^{|n'|}(u),u)) \\ &\equiv R(R^{|n'|}(u)) \quad (\text{because } |R^{|n'|}(u)| = |R^{|n'|-1}(u)| = \cdots = |u|) \\ &\equiv R^{|n'|+1}(u) = R^{|n|}(u). \end{aligned}$$

We next show that for every bitstrings u and n, $G(u,\mathtt{B}n) \equiv B(u) \bullet G(R(u),n)$. This is also proved by induction on $|n|$: when $|n| = 0$, i.e., $n = \mathtt{nil}$,

$$\begin{aligned} G(u,\mathtt{Bnil}) &\equiv \mathtt{rec}(\mathtt{nil}, \lambda m.\lambda r.r \bullet B(R'(u,m)), \mathtt{Bnil}) \\ &\equiv G(u,\mathtt{nil}) \bullet B(R'(u,\mathtt{nil})) \\ &\qquad (\text{because } G(u,\mathtt{nil}) \equiv \mathtt{rec}(\mathtt{nil}, \lambda m\,.\,\lambda r\,.\,r \bullet B(R'(u,m)), \mathtt{nil}) \equiv \mathtt{nil}) \\ &\equiv B(u) \;\equiv\; B(u) \bullet G(u,\mathtt{nil}); \end{aligned}$$

when $n \equiv \mathtt{B}n'$,

$$\begin{aligned} G(u,\mathtt{BB}n') &\equiv \mathtt{rec}(\mathtt{nil}, \lambda m.\lambda r.r \bullet B(R'(u,m)), \mathtt{BB}n') \\ &\equiv G(u,\mathtt{B}n') \bullet B(R'(u,\mathtt{B}n')) \\ &\equiv B(u) \bullet G(R(u),n') \bullet B(R^{|n'|+1}(u)) \\ &\equiv B(u) \bullet G(R(u),n') \bullet B(R'(R(u),n')). \end{aligned}$$

Because

$$\begin{aligned} G(R(u),\mathtt{B}n') &\equiv \mathtt{rec}(\mathtt{nil}, \lambda m.\lambda r.r \bullet B(R'(u,m)), \mathtt{B}n' \\ &\equiv G(R(u),n') \bullet B(R'(R(u),n')), \end{aligned}$$

it holds that

$$B(u)\bullet G(R(u),n')\bullet B(R'(R(u),n')) \equiv B(u)\bullet G(R(u),\mathtt{B}n'),$$

hence $G(u,\mathtt{B}n) \equiv B(u)\bullet G(R(u),n)$.

We next prove the computational indistinguishability between the two programs in the assertion:

$$\begin{aligned}
&\lambda x\,.\,\mathtt{bind}\ u = \boldsymbol{rs}(x)\ \mathtt{in}\ \mathtt{val}(G(u,\mathtt{B}n))\\
\equiv\ &\lambda x\,.\,\mathtt{bind}\ u = \boldsymbol{rs}(x)\ \mathtt{in}\ \mathtt{val}(B(u)\bullet G(R(u),n))\\
\equiv\ &\lambda x\,.\,\mathtt{bind}\ u = \boldsymbol{rs}(x)\ \mathtt{in}\ \mathtt{val}(\boldsymbol{hd}(g_1(u))\bullet G(\boldsymbol{tl}(g_1(u)),n))\\
\simeq\ &\lambda x\,.\,\mathtt{bind}\ u = \boldsymbol{rs}(\mathtt{B}x)\ \mathtt{in}\ \mathtt{val}(\boldsymbol{hd}(u)\bullet G(\boldsymbol{tl}(u),n))\\
&\quad\text{(by the rule } \textit{SUB} \text{ and because } \lambda x.\mathtt{bind}\ u = \boldsymbol{rs}(x)\ \mathtt{in}\ \mathtt{val}(g_1(u)) \simeq \lambda x.\boldsymbol{rs}(\mathtt{B}x))\\
\equiv\ &\lambda x\,.\,\mathtt{bind}\ (\,b = \mathtt{rand}, u = \boldsymbol{rs}(x)\,)\ \mathtt{in}\ \mathtt{val}(\boldsymbol{hd}(b\bullet u)\bullet G(\boldsymbol{tl}(b\bullet u),n))\\
&\quad\text{(by the rule } \textit{RS-CONCAT})\\
\equiv\ &\lambda x\,.\,\mathtt{bind}\ (\,b = \mathtt{rand}, u = \boldsymbol{rs}(x)\,)\ \mathtt{in}\ \mathtt{val}(b\bullet G(u,n)).
\end{aligned}$$

□

We next prove that, given a polynomial p, one can use G to construct easily a PRG with the expansion factor $|p|$, and the proof will be done in the system $\boldsymbol{\mathcal{C}}$.

Proposition 1 *For every well typed (length-sensitive) polynomial* $\vdash p : \Box\mathsf{Bits} \to \mathsf{Bits}$*,*

$$\lambda x\,.\,\mathtt{bind}\ u = \boldsymbol{rs}(x)\ \mathtt{in}\ \mathtt{val}(G(u,p(u))) \simeq \lambda x\,.\,\boldsymbol{rs}(p(x))$$

Proof. The proof follows the traditional hybrid technique, but is reformulated using rules of the system $\boldsymbol{\mathcal{C}}$. We define first a hybrid function H:

$$H \overset{\text{def}}{=} \lambda u_1\,.\,\lambda u_2\,.\,\lambda n\,.\,(u_2 - n)\bullet G(u_1,n).$$

H is well typed in SLR with the following assertion:

$$\vdash H : \Box\mathsf{Bits} \to \mathsf{Bits} \multimap \Box\mathsf{Bits} \to \mathsf{Bits}.$$

Firstly,

$$\begin{aligned}
&\lambda x\,.\,\mathtt{bind}\ u_1 = \boldsymbol{rs}(x)\ \mathtt{in}\ \mathtt{bind}\ u_2 = \boldsymbol{rs}(p(x))\ \mathtt{in}\ \mathtt{val}(H(u_1,u_2,\mathtt{nil}))\\
\equiv\ &\lambda x\,.\,\mathtt{bind}\ u_1 = \boldsymbol{rs}(x)\ \mathtt{in}\ \mathtt{bind}\ u_2 = \boldsymbol{rs}(p(x))\ \mathtt{in}\ \mathtt{val}((u_2 - \mathtt{nil})\bullet G(u_1,\mathtt{nil}))\\
\equiv\ &\lambda x\,.\,\mathtt{bind}\ u_1 = \boldsymbol{rs}(x)\ \mathtt{in}\ \mathtt{bind}\ u_2 = \boldsymbol{rs}(p(x))\ \mathtt{in}\ \mathtt{val}(u_2\bullet G(u_1,\mathtt{nil}))\\
&\quad\text{(by the rule } \textit{CUT})\\
\equiv\ &\lambda x\,.\,\mathtt{bind}\ u_1 = \boldsymbol{rs}(x)\ \mathtt{in}\ \mathtt{bind}\ u_2 = \boldsymbol{rs}(p(x))\ \mathtt{in}\ \mathtt{val}(u_2)\\
&\quad\text{(because } G(u_1,\mathtt{nil}) \equiv \mathtt{nil})\\
\equiv\ &\lambda x\,.\,\boldsymbol{rs}(p(x)).
\end{aligned}$$

Next, for all bitstrings u_1, u_2, n such that $|u_2| = |n|$,

$$H(u_1,u_2,n) \equiv (u_2 - n)\bullet G(u_1,n) \equiv \mathtt{nil}\bullet G(u_1,n) \equiv G(u_1,n),$$

hence,

$$\begin{aligned}
&\lambda x\,.\,\mathtt{bind}\ u_1 = \boldsymbol{rs}(x)\ \mathtt{in}\ \mathtt{bind}\ u_2 = \boldsymbol{rs}(p(x))\ \mathtt{in}\ \mathtt{val}(H(u_1,u_2,p(x)))\\
\equiv\ &\lambda x\,.\,\mathtt{bind}\ u_1 = \boldsymbol{rs}(x)\ \mathtt{in}\ \mathtt{bind}\ u_2 = \boldsymbol{rs}(p(x))\ \mathtt{in}\ \mathtt{val}(G(u_1,p(x)))\\
\equiv\ &\lambda x\,.\,\mathtt{bind}\ u_1 = \boldsymbol{rs}(x)\ \mathtt{in}\ \mathtt{val}(G(u_1,p(u_1))).
\end{aligned}$$

Because for every numeral i such that $|i(x)| < |p(x)|$ for any bitstring x,

$$\begin{aligned}
& \lambda x\,.\,\texttt{bind}\ (\,u_1 = \boldsymbol{rs}(x), u_2 = \boldsymbol{rs}(p(x))\,)\ \texttt{in}\ \texttt{val}(H(u_1, u_2, \mathtt{B}i(x))) \\
\equiv\ & \lambda x\,.\,\texttt{bind}\ (\,u_1 = \boldsymbol{rs}(x), u_2 = \boldsymbol{rs}(p(x))\,)\ \texttt{in}\ \texttt{val}((u_2 - \mathtt{B}i(x))\bullet G(u_1, \mathtt{B}i(x))) \\
\simeq\ & \lambda x\,.\,\texttt{bind}\ (\,b = \texttt{rand}, u_1 = \boldsymbol{rs}(x), u_2 = \boldsymbol{rs}(p(x))\,)\ \texttt{in}\ \texttt{val}((u_2 - \mathtt{B}i(x))\bullet b\bullet G(u_1, i(x))) \\
& \qquad \text{(by Lemma 14 and the rule } \textit{SUB}\text{)} \\
\equiv\ & \lambda x\,.\,\texttt{bind}\ (\,b = \texttt{rand}, u_1 = \boldsymbol{rs}(x), u_2 = \boldsymbol{rs}(p(x) - \mathtt{B}i(x))\,)\ \texttt{in}\ \texttt{val}(u_2\bullet b\bullet G(u_1, i(x))) \\
& \qquad \text{(by the rule } \textit{RS-CUT}\text{, as } |\mathtt{B}i(x)| = |i(x)| + 1 \leq |p(x)| = |u_2|\text{)} \\
\equiv\ & \lambda x\,.\,\texttt{bind}\ (\,u_1 = \boldsymbol{rs}(x), u_2 = \boldsymbol{rs}((p(x) - \mathtt{B}i(x))\bullet 1)\,)\ \texttt{in}\ \texttt{val}(u_2\bullet G(u_1, i(x))) \\
& \qquad \text{(by the rule } \textit{RS-CONCAT}\text{)} \\
\equiv\ & \lambda x\,.\,\texttt{bind}\ (\,u_1 = \boldsymbol{rs}(x), u_2 = \boldsymbol{rs}(p(x) - i(x))\,)\ \texttt{in}\ \texttt{val}(u_2\bullet G(u_1, i(x))) \\
& \qquad \text{(because } |(p(x) - \mathtt{B}i(x))\bullet 1| = |p(x) - i(x)| - 1 + 1 = |p(x) - i(x)|\text{)} \\
\equiv\ & \lambda x\,.\,\texttt{bind}\ (\,u_1 = \boldsymbol{rs}(x), u_2 = \boldsymbol{rs}(p(x))\,)\ \texttt{in}\ \texttt{val}((u_2 - i(x))\bullet G(u_1, i(x))) \\
& \qquad \text{(by the rule } \textit{RS-CUT}\text{)} \\
\equiv\ & \lambda x\,.\,\texttt{bind}\ (\,u_1 = \boldsymbol{rs}(x), u_2 = \boldsymbol{rs}(p(x))\,)\ \texttt{in}\ \texttt{val}(H(u_1, u_2, i(x)))
\end{aligned}$$

by the rule *H-IND*,

$$\begin{aligned}
& \lambda x\,.\,\texttt{bind}\ (\,u_1 = \boldsymbol{rs}(x), u_2 = \boldsymbol{rs}(p(x))\,)\ \texttt{in}\ \texttt{val}(H(u_1, u_2, p(x))) \\
\simeq\ & \lambda x\,.\,\texttt{bind}\ (\,u_1 = \boldsymbol{rs}(x), u_2 = \boldsymbol{rs}(p(x))\,)\ \texttt{in}\ \texttt{val}(H(u_1, u_2, \texttt{nil})),
\end{aligned}$$

i.e., $\lambda x\,.\,\texttt{bind}\ u = \boldsymbol{rs}(x)\ \texttt{in}\ \texttt{val}(G(u, p(x))) \simeq \lambda x\,.\,\boldsymbol{rs}(p(x))$. □

Theorem 5. *The CSLR term $\lambda x\,.\,G(x, p(x))$ is a pseudorandom generator with the expansion factor $|p|$.*

Proof. Obviously from Proposition 1 and Definition 2. □

4.2 Relating pseudorandomness and next-bit unpredictability

Our second example is the equivalence between pseudorandomness and next-bit unpredictability. We first reformulate the notion of next-bit unpredictability in CSLR: a positive polynomial f such that $\vdash f : \Box\mathsf{Bits} \rightarrow \mathsf{Bits}$ is *next-bit unpredictable* if for all canonical numeral i such that $|i| < |f|$,

$$\begin{array}{lcl}
\lambda x\,.\ \texttt{bind}\ u = \boldsymbol{rs}(x)\ \texttt{in} & & \lambda x\,.\ \texttt{bind}\ u = \boldsymbol{rs}(x)\ \texttt{in}\ \texttt{bind}\ b = \texttt{rand}\ \texttt{in} \\
\qquad \texttt{val}(\boldsymbol{pref}(f(u), \mathtt{B}_1 i(x))) & \simeq & \qquad \texttt{val}(\boldsymbol{pref}(f(u), i(x))\bullet b)
\end{array}\ .$$

Lemma 15. *Pseudorandomness implies next-bit unpredictability: if a positive polynomial f is a pseudorandom generator, then it is next-bit unpredictable.*

Proof. Because f is a pseudorandom generator,

$$\lambda x.\texttt{bind}\ u = \boldsymbol{rs}(x)\ \texttt{in}\ \texttt{val}(f(u)) \simeq \lambda x.\boldsymbol{rs}(f(x)).$$

Hence,

$$\begin{aligned}
&\lambda x\,.\,\mathtt{bind}\ u = \boldsymbol{rs}(x)\ \mathtt{in}\ \mathtt{val}(\boldsymbol{pref}(f(u), \mathtt{B}_1 i))\\
\simeq\ &\lambda x\,.\,\mathtt{bind}\ u = \boldsymbol{rs}(f(x))\ \mathtt{in}\ \mathtt{val}(\boldsymbol{pref}(u, \mathtt{B}_1 i))\\
&\qquad \text{(because } f \text{ is a pseudorandom generator)}\\
\equiv\ &\lambda x\,.\,\boldsymbol{rs}(\boldsymbol{pref}(f(x), \mathtt{B}_1 i)) \qquad \text{(by the rule } \textit{RS-SPLIT}\text{)}\\
\equiv\ &\lambda x\,.\,\boldsymbol{rs}(\mathtt{B}_1 \boldsymbol{pref}(f(x), i)) \qquad \text{(by the rule } \textit{RS-NEXT-BIT}\text{)}\\
\equiv\ &\lambda x\,.\,\mathtt{bind}\ b = \mathtt{rand}\ \mathtt{in}\ \mathtt{bind}\ u = \boldsymbol{rs}(\boldsymbol{pref}(f(x), i))\ \mathtt{in}\ \mathtt{val}(b \bullet u)\\
&\qquad \text{(by the definition of } \boldsymbol{rs}\text{)}\\
\equiv\ &\lambda x\,.\,\mathtt{bind}\ b = \mathtt{rand}\ \mathtt{in}\ \mathtt{bind}\ u = \boldsymbol{rs}(\boldsymbol{pref}(f(x), i))\ \mathtt{in}\ \mathtt{val}(u \bullet b)\\
&\qquad \text{(by the rule } \textit{RS-COMMUT}\text{)}\\
\equiv\ &\lambda x\,.\,\mathtt{bind}\ b = \mathtt{rand}\ \mathtt{in}\ \mathtt{bind}\ u = \boldsymbol{rs}(x)\ \mathtt{in}\ \mathtt{val}(\boldsymbol{pref}(f(u), i) \bullet b)\\
&\qquad \text{(by the rule } \textit{RS-SPLIT}\text{)}
\end{aligned}$$

Note that in the above proof i is a function and we omit the argument, but this is careful because

Lemma 16. *Next-bit unpredictability implies pseudorandomness: if a positive polynomial f is next-bit unpredictable, then it is a pseudorandom generator with expansion $|f|$.*

Proof. The proof also uses the hybrid technique. We define the hybrid function as follows:

$$H \stackrel{\text{def}}{=} \lambda x.\lambda y.\lambda z.\boldsymbol{pref}(f(x), z) \bullet \boldsymbol{suff}(y, z).$$

It can be easily proved that, for all bitstrings u, v such that $|v| = |f(u)|$, $H(u, v, \mathtt{nil}) \equiv v$ and $H(u, v, f(u)) \equiv f(u)$, hence

$$\begin{aligned}
&\lambda x\,.\,\mathtt{bind}\ (\,u = \boldsymbol{rs}(x), v = \boldsymbol{rs}(f(x))\,)\ \mathtt{in}\ \mathtt{val}(H(u, v, \mathtt{nil})) \equiv \boldsymbol{rs}(f(x))\\
&\lambda x\,.\,\mathtt{bind}\ (\,u = \boldsymbol{rs}(x), v = \boldsymbol{rs}(f(x))\,)\ \mathtt{in}\ \mathtt{val}(H(u, v, f(x))) \equiv \lambda x\,.\,\mathtt{bind}\ u = \boldsymbol{rs}(x)\ \mathtt{in}\ \mathtt{val}(f(u)).
\end{aligned}$$

We then prove the hybrid step: for all canonical polynomial i such that $|i| < |f|$,

$$\begin{aligned}
&\lambda x\,.\,\mathtt{bind}\ (\,u = \boldsymbol{rs}(x), v = \boldsymbol{rs}(f(x))\,)\ \mathtt{in}\ \mathtt{val}(H(u, v, \mathtt{B}_1 i))\\
\equiv\ &\lambda x\,.\,\mathtt{bind}\ (\,u = \boldsymbol{rs}(x), v = \boldsymbol{rs}(f(x))\,)\ \mathtt{in}\ \mathtt{val}(\boldsymbol{pref}(f(u), \mathtt{B}_1 i) \bullet \boldsymbol{suff}(v, \mathtt{B}_1 i))\\
\simeq\ &\lambda x\,.\,\mathtt{bind}\ (\,u = \boldsymbol{rs}(x), b = \mathtt{rand}, v = \boldsymbol{rs}(f(x))\,)\ \mathtt{in}\ \mathtt{val}(\boldsymbol{pref}(f(u), i) \bullet b \bullet \boldsymbol{suff}(v, \mathtt{B}_1 i))\\
&\qquad \text{(because } f \text{ is next-bit unpredictable)}\\
\equiv\ &\lambda x\,.\,\mathtt{bind}\ (\,u = \boldsymbol{rs}(x), b = \mathtt{rand}, v = \boldsymbol{rs}(\boldsymbol{suff}(f(x), \mathtt{B}_1 i))\,)\ \mathtt{in}\ \mathtt{val}(\boldsymbol{pref}(f(u), i) \bullet b \bullet v)\\
&\qquad \text{(by the rule } \textit{RS-SPLIT}\text{)}\\
\equiv\ &\lambda x\,.\,\mathtt{bind}\ (\,u = \boldsymbol{rs}(x), v = \boldsymbol{rs}(1 \bullet \boldsymbol{suff}(f(x), \mathtt{B}_1 i))\,)\ \mathtt{in}\ \mathtt{val}(\boldsymbol{pref}(f(u), i) \bullet v)\\
&\qquad \text{(by the rule } \textit{RS-CONCAT}\text{)}\\
\equiv\ &\lambda x\,.\,\mathtt{bind}\ (\,u = \boldsymbol{rs}(x), v = \boldsymbol{rs}(\boldsymbol{suff}(f(x), i))\,)\ \mathtt{in}\ \mathtt{val}(\boldsymbol{pref}(f(u), i) \bullet v)\\
&\qquad \text{(by the rule } \textit{RS-EQUIV} \text{ since } |1 \bullet \boldsymbol{suff}(f(x), \mathtt{B}_1 i)| = |\boldsymbol{suff}(f(x), i)|\text{)}\\
\equiv\ &\lambda x\,.\,\mathtt{bind}\ (\,u = \boldsymbol{rs}(x), v = \boldsymbol{rs}(f(x))\,)\ \mathtt{in}\ \mathtt{val}(\boldsymbol{pref}(f(u), i) \bullet \boldsymbol{suff}(v, i))\\
&\qquad \text{(by the rule } \textit{RS-SPLIT}\text{)}\\
\equiv\ &\lambda x\,.\,\mathtt{bind}\ (\,u = \boldsymbol{rs}(x), v = \boldsymbol{rs}(f(x))\,)\ \mathtt{in}\ \mathtt{val}(H(u, v, i)).
\end{aligned}$$

Hence, by the rule *H-IND*,

$$\lambda x \,.\, \mathtt{bind}\; u = \boldsymbol{rs}(x)\; \mathtt{in}\; \mathtt{val}(f(u)) \equiv \lambda x \,.\, \boldsymbol{rs}(f(x)),$$

i.e., f is a pseudorandom generator with expansion $|f|$. □

Theorem 6. *A positive polynomial is a pseudorandom generator if and only if it is next-bit unpredictable.*

Proof. The two directions are proved respectively in the above two lemmas. □

5 Related work

Many researchers in cryptography have realized that the increasing complexity of cryptographic proofs is now an obstacle that cannot be ignored and formal techniques must be introduced to write and check cryptographic proofs. Some proof systems similar to ours have been proposed in recent years.

The PPC (probabilistic polynomial-time process calculus) system designed by Mitchell et al. [11] is based on a variant of CCS with bound replication and messages that are computable in probabilistic polynomial-time. An equational proof system is also given in their system to prove the observational equivalence between processes, and the soundness is established upon a form of probabilistic bisimulation. Interestingly, they mention that terms (or messages) in their language can be those of OSLR (the probabilistic extension of SLR), but we are not clear how much expressitivity PPC achieves by adding the process part. It is probably more natural for modeling protocols, but no such examples are given in their paper.

Impagliazzo and Kapron have proposed two logic systems for reasoning about cryptographic constructions [9]. Their first logic is based on a non-standard arithmetic model, which they prove captures probabilistic polynomial-time computations. While it is a complex and general system, they define a simpler logic on top of the first one, with rules justifying computational indistinguishability. The language in their second logic is very close to a functional language but is unfortunately not precisely defined, and in fact, this leads to a subtle flaw in their proofs using the logic: the *SUB* rule in their logic requires that the substitute programs must be closed terms, but this is not respected in their proofs. In particular, the hybrid proofs often have a program of the form $\mathtt{let}\; i \leftarrow p(\boldsymbol{n})\; \mathtt{in}\; e$, where e has a free variable x and it is often substituted by indistinguishable programs, but, for instance, if the two programs also have a bound variable i receiving a random number:

$$\mathtt{let}\; i \leftarrow p(\boldsymbol{n})\; \mathtt{in}\; e_1 \simeq \mathtt{let}\; i \leftarrow p(\boldsymbol{n})\; \mathtt{in}\; e_1,$$

according to the rule *SUB* we can only deduce

$$\mathtt{let}\; i \leftarrow p(\boldsymbol{n})\; \mathtt{in}\; e[\mathtt{let}\; i \leftarrow p(\boldsymbol{n})\; \mathtt{in}\; e_1/x] \simeq \mathtt{let}\; i \leftarrow p(\boldsymbol{n})\; \mathtt{in}\; e[\mathtt{let}\; i \leftarrow p(\boldsymbol{n})\; \mathtt{in}\; e_2/x],$$

but never

$$\mathtt{let}\; i \leftarrow p(\boldsymbol{n})\; \mathtt{in}\; e[e_1/x] \simeq \mathtt{let}\; i \leftarrow p(\boldsymbol{n})\; \mathtt{in}\; e[e_2/x].$$

However, the latter is used in many proofs in [9]. Furthermore, they claim that by introducing rules justifying directly the computational indistinguishability between programs, they avoid explicit reasoning about the probability, but the rule *UNIV* contains a premise in their base logic (in the arithmetic model) and proving that might still involve reasoning about the probability.

In our knowledge, both the proof systems in PPC and the IK-logic have not been automated. Meanwhile, Nowak has proposed a framework for formal verification of cryptographic primitives and it has been implemented in the proof-assistant Coq [13]. It is in fact a formalization of the game-based security proofs, an approach advocated by several researchers in cryptography [4, 15], where proofs are done by generating a sequence of games and transformations between games must be proved computationally sound. In Nowak's formalization, games are seen as syntactic objects and game transformations are syntactic manipulations that can be verified in the proof-assistant, but the complexity-theoretic issues are not considered. Similar work include the system by Barthe et al., also implemented in Coq but using an imperative language [2] and the other one by Backes et al., implemented in Isabelle/HOL and using a functional language with references and events [1].

Blanchet's CryptoVerif is another automated tool supporting game-based cryptographic proofs, but not based on any existing theorem provers [5]. Unlike previously mentioned work, CryptoVerif aims at generating the sequence of games based on a collection of predefined transformations, instead of verifying the computational soundness of transformations defined by users.

6 Conclusion

We present an equational proof system that can be used to prove the computational indistinguishability between programs, and have proved that rules in the system are sound with respect to the set-theoretic semantics, hence the standard notion of security. We also show that the system is applicable in cryptography by using it to verify a cryptographic construction of pseudorandom generator.

Unlike the related work mentioned in the previous section, where they either define a language from scratch or do not give a precise language definition, our language is extended from Hofmann's SLR, which has a very solid mathematical support based on Bellantoni and Cook's safe recursion and a nice mechanism for the characterization of polynomial-time computations. Cryptographers may argue that examples given in the paper are experimental, but we believe that the system can be used to verify more realistic cryptographic constructions. In particular, one should be able to formulate the game-based approach in our system without much difficulty. Furthermore, as higher-order functions are already native in the language, we also believe that the system can be used to verify cryptographic protocols in the computational model, but possibly we need additional mechanisms like references in [1] to keep state between invocations of oracles.

References

1. M. Backes, M. Berg, and D. Unruh. A formal language for cryptographic pseudocode. In *4th Workshop on Formal and Computational Cryptography (FCC 2008)*, 2008.
2. G. Barthe, B. Grégoire, R. Janvier, and S. Zanella Béguelin. Formal certification of code-based cryptographic proofs. In *4th Workshop on Formal and Computational Cryptography (FCC 2008)*, 2008.
3. Stephen Bellantoni and Stephen A. Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.
4. M. Bellare and P. Rogaway. Code-based game-playing proofs and the security of triple encryption. Cryptology ePrint Archive, Report 2004/331, 2004.
5. Bruno Blanchet. A computationally sound mechanized prover for security protocols. In *IEEE Symposium on Security and Privacy (S&P'06)*, pages 140–154, 2006.
6. Oded Goldreich. *The Foundations of Cryptography: Basic Tools*. Cambridge University Press, 2001.
7. Martin Hofmann. A mixed modal/linear lambda calculus with applications to bellantoni-cook safe recursion. In *Proceedings of the International Workshop of Computer Science Logic (CSL'97)*, volume 1414 of *LNCS*, pages 275–294. Springer, 1998.

8. Martin Hofmann. Safe recursion with higher types and BCK-algebra. *Annals of Pure and Applied Logic*, 104(1-3):113–166, 2000.
9. Russell Impagliazzo and Bruce M. Kapron. Logics for reasoning about cryptographic constructions. *Journal of Computer and System Sciences*, 72(2):286–320, 2006.
10. John C. Mitchell, Mark Mitchell, and Andre Scedrov. A linguistic characterization of bounded oracle computation and probabilistic polynomial time. In *39th Annual Symposium on Foundations of Computer Science (FOCS'98)*, pages 725–733, 1998.
11. John C. Mitchell, Ajith Ramanathan, Andre Scedrov, and Vanessa Teague. A probabilistic polynomial-time process calculus for the analysis of cryptographic protocols. *Theoretical Computer Science*, 353(1-3):118–164, 2006.
12. Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.
13. David Nowak. A framework for game-based security proofs. In *9th International Conference of Information and Communications Security (ICICS 2007)*, volume 4861 of *LNCS*, pages 319–333. Springer, 2008.
14. Norman Ramsey and Avi Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02)*, pages 154–165, 2002.
15. Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332, 2004.