Under consideration for publication in Math. Struct. in Comp. Science

The computational SLR: a logic for reasoning about computational indistinguishability

Yu Zhang

Laboratory for Computer Science, Institute of Software, Chinese Academy of Sciences

P. O. Box 8718, Beijing 100190, China

yzhang@ios.ac.cn

Received 2 December 2009

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 verification of cryptographic protocols in recent years has switched its focus from the symbolic 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 ultimate 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 (Hof98;

Hof00), a functional programming language that implements Bellantoni and Cook's safe recursion (BC92). 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. (MMS98), 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 equational proof 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 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 to illustrate its usability in cryptography. Section 5 summarizes related work and Section 6 concludes the paper.

2. The computational SLR

We start by defining 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 (BC92). This is an intrinsic, purely syntactic mechanism: variables are divided into 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 (Hof98; Hof00). 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 traditional definition of the computational indistinguishability in our language.

2.1. *The non-polymorphic SLR for bitstrings*

Types are defined by:

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

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

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

Fig. 1. Sub-typing rules for SLR

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. SLR uses aspects to represent these function spaces: $\tau \xrightarrow{a} \tau'$ is a function type with aspect a, which is either $\mathfrak{m} = (\text{modal}, \text{nonlinear})$ for $\Box\tau \to \tau'$, or $\mathfrak{n} = (\text{nonmodal}, \text{nonlinear})$ for $\tau \to \tau'$, or $\mathfrak{l} = (\text{nonmodal}, \text{linear})$ for $\tau \multimap \tau'$. The aspects are ordered: $\mathfrak{m} \leq \mathfrak{n} \leq \mathfrak{l}$. They are also used to tag variables in typing contexts.

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.

Expressions of SLR are defined by the following grammar:

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

$\mathtt{B}_0$ 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 semantically 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: $\mathtt{case}_\tau(n, \langle e, f_0, f_1 \rangle)$ 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 as well as aspects to variables. A context is typically written as a list of bindings

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

$$\frac{\Gamma, \Delta_1 \vdash e_1 : \tau \xrightarrow{a} \tau' \quad \Gamma, \Delta_2 \vdash e_2 : \tau \quad \Gamma \text{ nonlinear} \quad x :^{a'} \sigma \in \Gamma, \Delta_2 \text{ implies } a' \leq a}{\Gamma, \Delta_1, \Delta_2 \vdash e_1 e_2 : \tau'}\ \textit{T-APP}$$

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

$$\frac{\Gamma, \Delta_1 \vdash e_1 : \tau_1 \quad \Gamma, \Delta_2 \vdash e_2 : \tau_2 \quad \Gamma \text{ nonlinear}}{\Gamma, \Delta_1, \Delta_2 \vdash e_1 \otimes e_2 : \tau_1 \otimes \tau_2}\ \textit{T-TENSOR}$$

$$\frac{\Gamma, \Delta_1, x :^{\mathfrak{l}} \tau_1, y :^{\mathfrak{l}} \tau_2 \vdash e : \tau \quad \Gamma, \Delta_2 \vdash e' : \tau_1 \otimes \tau_2 \quad \Gamma \text{ nonlinear}}{\Gamma, \Delta_1, \Delta_2 \vdash \mathtt{let}\ x \otimes y = e'\ \mathtt{in}\ e : \tau}\ \textit{T-LET}$$

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

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

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

Fig. 2. Typing rules of SLR

$x_1 :^{a_1} \tau_1, \ldots, x_n :^{a_n} \tau_n$, 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 (MMS98). However, OSLR is a functional language with side-effects, which means that the value of a program depends on the evaluation strategy (call-by-name or call-by-value), which makes it difficult to deal with substitution when we build a logic upon the language. Hence we adopt a different syntax from Moggi's computational λ-calculus (Mog91), a pure functional language 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 terminates and can be probabilistic, hence it will return one of a set of values, each with

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

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

Fig. 3. Additional typing rules of the computational SLR

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:

$e_1, e_2, \ldots$	::=	$\ldots$	SLR terms
	|	$\texttt{rand}$	oracle bit
	|	$\texttt{val}(e)$	deterministic computation
	|	$\texttt{bind}\ x = e_1\ \texttt{in}\ e_2$	sequential computation

The constant `rand` returns a random bit 0 or 1, each with the probability $\frac{1}{2}$. $\texttt{val}(e)$ is the trivial (deterministic) computation which returns e with the probability 1. $\texttt{bind}\ x = e_1\ \texttt{in}\ e_2$ 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 $\texttt{bind}\ x_1 = e_1\ \texttt{in}\ \ldots \texttt{bind}\ x_n = e_n\ \texttt{in}\ e$ as $\texttt{bind}\ (\, x_1 = e_1, \ldots, x_n = e_n \,)\ \texttt{in}\ e$. 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 its type does not have monadic components. For instance, the function $\lambda x^{\mathsf{Bits}} . (\lambda y^{\mathsf{TBits}} . x)\texttt{rand}$ 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*.

As in many 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:

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

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:

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

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 (RP02): 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 (RP02) 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 (MMS98), 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 set-theoretic model does not distinguish between the normal products and tensor products, neither between the three sorts of function spaces.

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

Theorem 1 (Hofmann (Hof00)). The set-theoretic interpretations of closed terms of type $\Box\mathsf{Bits} \to \mathsf{Bits}$ in SLR define exactly polynomial-time computable functions.

It is worth mentioning that tensor products and linear functions in SLR, which leads to a relatively complex type-checking that is likely not an easy job for cryptographers, are not necessary for capturing PTIME computations when we do not consider high-order recusions, which can often ease the programming in SLR. In practice, a simpler language without high-order recusion (consequently no need for tensor products and linear functions) is probably enough for cryptographic use, but we keep the language as it is for a comprehensive system.

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 model that they use to prove the complexity result can be also used to interpret CSLR. 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 $\Box\mathsf{Bits} \to \mathsf{TBits}$. This permits us to reuse the result of (MMS98), adapted for CSLR:

Theorem 2 (Mitchell et al. (MMS98)). The set-theoretic interpretations of closed terms of type $\Box\mathsf{Bits} \to \mathsf{TBits}$ 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 $\Box\mathsf{Bits} \to \mathsf{Bits}$) 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

Interpretation of types:

$$
\begin{array}{lcl}
[\![\mathsf{Bits}]\!] & = & \mathbb{B} \\
[\![\tau \times \tau']\!] & = & [\![\tau]\!] \times [\![\tau']\!] \\
[\![\tau \otimes \tau']\!] & = & [\![\tau]\!] \times [\![\tau']\!] \\
\left[\!\!\left[\tau \xrightarrow{a} \tau'\right]\!\!\right] & = & [\![\tau]\!] \to [\![\tau']\!] \\
[\![\mathsf{T}\tau]\!] & = & \mathcal{D}_{[\![\tau]\!]}
\end{array}
$$

Interpretation of terms:

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

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

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 $\Box\mathsf{Bits} \to \tau$) 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 bitstrings containing bit 1 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 (Gol01, Definition 3.2.2) in the setting of CSLR.

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

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

where ϵ denotes the empty bitstring.

Here w is usually considered as the security parameter in a cryptographic context.

Note that the above definition is a more general than that of (Gol01). Here we consider programs that can return values of an arbitrary type, while in the definition of (Gol01), computational indistinguishability is only defined for distributions of bitstrings. One may argue that for reasoning about crypto-systems, higher-order types are not necessary, but the general definition turns out to be very helpful for formalizing game-based proofs in our recent investigation.

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 $\boldsymbol{rs}$:

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

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

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

$\boldsymbol{rs}$ 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 $\vdash \boldsymbol{rs} : \Box\mathsf{Bits} \to \mathsf{TBits}$.

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 $\boldsymbol{conc}$:

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

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

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

$h_{\boldsymbol{conc}}$ 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 $\boldsymbol{conc}$ 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***:

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

Tail function ***tl***:

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

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

— Split function ***split***:

$$\boldsymbol{split} \stackrel{\text{def}}{=} \lambda x \,.\, \lambda n \,.\, \mathtt{rec}(\mathtt{nil} \otimes x, h_{\boldsymbol{split}}, n),$$

where

$$h_{\boldsymbol{split}} \stackrel{\text{def}}{=} \lambda m \,.\, \lambda r \,.\quad \begin{array}{l} \mathtt{let}\ v_1 \otimes v_2 = r\ \mathtt{in} \\ \mathtt{case}(v_2, \langle v_1 \otimes v_2, \lambda y.(v_1 \bullet 0) \otimes y, \lambda y.(v_1 \bullet 1) \otimes y \rangle). \end{array}$$

$\boldsymbol{split}(x, n)$ 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:

$$\begin{array}{l} \boldsymbol{pref} \stackrel{\text{def}}{=} \lambda x \,.\, \lambda n \,.\, \mathtt{let}\ u_1 \otimes u_2 = \boldsymbol{split}(x, n)\ \mathtt{in}\ u_1, \\ \boldsymbol{suff} \stackrel{\text{def}}{=} \lambda x \,.\, \lambda n \,.\, \mathtt{let}\ u_1 \otimes u_2 = \boldsymbol{split}(x, n)\ \mathtt{in}\ u_2. \end{array}$$

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

— Cut function ***cut***:

$$\boldsymbol{cut} \stackrel{\text{def}}{=} \lambda x \,.\, \lambda n \,.\, \boldsymbol{pref}(x, \boldsymbol{suff}(x, n)).$$

$\boldsymbol{cut}(x, n)$ 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 on top of CSLR, through which one can justify the computational indistinguishability between CSLR programs at the syntactic level.

The proof system contains two sets of rules: the first set (Figure 5) are rules for justifying semantic equivalence between CSLR programs (we write $e_1 \equiv e_2$ if e_1, e_2 are semantically equivalent), and the second set (Figure 6) are rules for justifying computational indistinguishability.

The first set are standard rules in typed λ-calculi, with axioms (monad laws) for probabilistic computations. Rules in the second set are similar as in the logic of Impagliazzo and Kapron (IK06) (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 we do not have the *EDIT* rule for managing bitstrings, as appears internally in their logic, because there is no primitive operations in CSLR for editing bitstrings except the two bit constructors $\mathtt{B}_0, \mathtt{B}_1$. Many bitstring operations are defined as CSLR functions and we introduce a series of lemmas (see 3.2), which can be used in proofs in the same way as system rules. By

Axioms:

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

$$\frac{}{\texttt{rec}(e_1, e_2, \texttt{B}e) \equiv e_2(e, \texttt{rec}(e_1, e_2, e))}\ \textit{AX-REC} \qquad \frac{x \notin FV(e')}{\texttt{case}(e, e', \lambda x.e', \lambda x.e') \equiv e'}\ \textit{AX-CASE}$$

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

$$\frac{y \notin FV(e)}{\lambda x.e \equiv \lambda y.e[y/x]}\ \textit{AX-}\alpha \qquad \frac{}{(\lambda x.e)e' \equiv e[e'/x]}\ \textit{AX-}\beta \qquad \frac{x \notin FV(e)}{\lambda x.ex \equiv e}\ \textit{AX-}\eta$$

$$\frac{i = 1, 2}{\texttt{proj}_i\langle e_1, e_2\rangle \equiv e_i}\ \textit{AX-PROJ-i} \qquad \frac{}{\langle \texttt{proj}_1 e, \texttt{proj}_2 e\rangle \equiv e}\ \textit{AX-PAIR}$$

$$\frac{}{\texttt{let}\ x_1 \otimes x_2 = e_1 \otimes e_2\ \texttt{in}\ e \equiv e[e_1/x_1, e_2/x_2]}\ \textit{AX-LET}$$

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

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

$$\frac{y \notin FV(e')}{\texttt{bind}\ x = e\ \texttt{in}\ e' \equiv \texttt{bind}\ y = e\ \texttt{in}\ e'}\ \textit{AX-BIND-}\alpha \qquad \frac{x \notin FV(e')}{\texttt{bind}\ x = e\ \texttt{in}\ e' \equiv e'}\ \textit{AX-BIND-}\eta$$

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

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

Inference rules:

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

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

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

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

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

Fig. 5. Rules for semantic equivalence

$$\frac{\vdash e_i : \Box\mathsf{Bits} \to \tau \ (i = 1, 2) \quad e_1 \equiv e_2}{e_1 \simeq e_2}\ \textit{EQUIV}$$

$$\frac{\vdash e_i : \Box\mathsf{Bits} \to \tau \ (i = 1, 2, 3) \quad e_1 \simeq e_2 \quad e_2 \simeq e_3}{e_1 \simeq e_3}\ \textit{TRANS-INDIST}$$

$$\frac{x :^{\mathsf{n}} \mathsf{Bits}, y :^{\mathsf{n}} \tau \vdash e : \tau' \quad \vdash e_i : \Box\mathsf{Bits} \to \tau \ (i = 1, 2) \quad e_1 \simeq e_2}{\lambda x \,.\, e[e_1(x)/y] \simeq \lambda x \,.\, e[e_2(x)/y]}\ \textit{SUB}$$

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

Fig. 6. Rules for computational indistinguishability

doing so we avoid the complexity analysis regarding bitstring operations that appears in the IK-logic, since all bitstring operations in CSLR are guaranteed to be polynomial-time computable by the typing system.

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.

We remark that the rule *TRANS-INDIST* is safe and will not break the complexity constraint, because the number of times of applying a rule in a proof is irrelvant to the security parameter of the programs under testing.

3.1. *Soundness of the proof system*

To show that the CSLR proof system is *sound* with respect to the set-theoretic semantics, we prove the soundness of the two sets of rules.

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 the CSLR proof system, 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 CSLR proof system, then e_1 and e_2 are computationally indistinguishable.

Proof. We prove that rules in Figure 6 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 $|w| \geq n$,

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

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

To prove the soundness of the rule *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 $f_i = \lambda x.e[e_i(w)/y]$ $(i = 1, 2)$ are the two programs in the conclusion part of the rule *SUB*. We then construct another adversary $\mathcal{A}'$:

$$\mathcal{A}' \stackrel{\text{def}}{=} \lambda z\,.\,\lambda z'\,.\,\mathcal{A}(z, e[z/x, z'/y]),$$

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

$$[\![\mathcal{A}'(w, e_i(w))]\!] = [\![\mathcal{A}(w, e[w/x, e_i(w)/y])]\!] = [\![\mathcal{A}(w, (\lambda x.e[e_i(x)/y])w)]\!],\ (i = 1, 2)$$

hence

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

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 that of TRANS-INDIST. Let $\mathcal{A}$ be an arbitrary well-typed adversary and q be an arbitrary positive polynomial. Define another polynomial: $q' \stackrel{\text{def}}{=} \lambda x\,.\,\mathtt{rec}(\mathtt{nil}, \lambda m.\lambda r.q'(x) \bullet r, p(x))$. Clearly, for all bitstrings u, $|q'(u)| = |q(u)| \cdot |p(u)|$. Because $\lambda x.e[i(x)/n] \simeq \lambda x.e[\mathtt{B}i(x)/n]$ 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,

$$|\mathbf{Pr}[\epsilon \leftarrow [\![\mathcal{A}(w, e[\mathtt{nil}/n])]\!]] - \mathbf{Pr}[\epsilon \leftarrow [\![\mathcal{A}(w, e[1/n])]\!]]| < \frac{1}{|q'(w)|}$$
$$\ldots\ldots$$
$$|\mathbf{Pr}[\epsilon \leftarrow [\![\mathcal{A}(w, e[p(w)-1/n])]\!]] - \mathbf{Pr}[\epsilon \leftarrow [\![\mathcal{A}(w, e[p(w)/n])]\!]]| < \frac{1}{|q'(w)|}.$$

Therefore,

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

and according to Definition 1, $\lambda x.e[\mathtt{nil}/n] \simeq \lambda x.e[p(x)/n]$. □

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 lemmas are not internal rules of the proof system, but we shall name and use them as if they are.

A large part of proofs can be done in the CSLR proof system. In the paper we only give a few of them as examples — others are left as exercises for interested readers.

Lemma 1. For every bitstring u, the functions $\lambda x.\boldsymbol{split}(u, x)$, $\lambda x.\boldsymbol{pref}(u, x)$, $\lambda x.\boldsymbol{suff}(u, x)$ 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

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

□

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 $\textbf{\textit{split}}(u, u') \equiv u_1 \otimes u_2$ 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:

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

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

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

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

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

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 $|u'| \geq |u|$,

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

Proof. Firstly, for every bitstring u,

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

Because

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

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

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

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

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

Proof. The proof is omitted. □

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

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

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 proof is omitted. □

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

Proof. The proof is omitted. □

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

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

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

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

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

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

then

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

□

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

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

Proof. The proof is omitted. □

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

Proof. The proof is omitted. □

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

Proof. The proof is omitted. □

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

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

Proof. The proof is omitted. □

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

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

Proof. The proof is omitted. □

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

$$\textbf{\textit{rs}}(\textbf{\textit{pref}}(u, \texttt{B}i)) \equiv \textbf{\textit{rs}}(\texttt{B}\textbf{\textit{pref}}(u, i)).$$

Proof. The proof is omitted. □

4. Cryptographic examples

In this section we give two cryptographic examples to illustrate the usability of the CSLR proof system in cryptography, by doing the security proofs in the proof system.

4.1. *Pseudorandom generators*

The first example verifies the correctness of Goldreich and Micali's construction of pseudorandom generator (Gol01). This example also appears in (IK06), but their proof has a subtle flaw (see Section 5 for explanation).

We first reformulate in CSLR the standard definition of pseudorandom generator (Gol01, Definition 3.3.1).

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

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

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

We recall the construction of Goldreich and Micali (Gol01) (reformulated in CSLR): Suppose that g_1 is a PRG with the expansion factor $|g_1|(x) = x + 1$, i.e.,

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

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

$$B \stackrel{\text{def}}{=} \lambda x\,.\,\textbf{\textit{hd}}(g_1(x)), \qquad\qquad R \stackrel{\text{def}}{=} \lambda x\,.\,\textbf{\textit{tl}}(g_1(x)).$$

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

$$G \stackrel{\text{def}}{=} \lambda u\,.\,\lambda n\,.\,\texttt{rec}(\texttt{nil}, \lambda m\,.\,\lambda r\,.\,r \bullet B(R'(u, m)), n),$$

where the function R' is defined as:

$$R' \stackrel{\text{def}}{=} \lambda u\,.\,\lambda n\,.\,\mathtt{rec}(u, \lambda m\,.\,\lambda r\,.\,R(\textbf{\textit{pref}}\,(r,u)), n).$$

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,

$$\lambda x\,.\,\mathtt{bind}\,u = \textbf{\textit{rs}}(x)\,\mathtt{in}\,\mathtt{val}(G(u,\mathtt{B}n)) \;\simeq\; \lambda x\,.\,\mathtt{bind}\,(\,b = \mathtt{rand}, u = \textbf{\textit{rs}}(x)\,)\,\mathtt{in}\,\mathtt{val}(b\bullet G(u,n)).$$

Proof. Because $R \stackrel{\text{def}}{=} \lambda x\,.\,\textbf{\textit{tl}}(g_1(x))$, 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., $n = \mathtt{nil}$,

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

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

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

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:

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

□

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 is done in the CSLR proof system.

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

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

Proof. The proof follows the traditional hybrid technique, but is reformulated using rules of the CSLR proof system. We define first a hybrid function H:

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

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,

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

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

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

hence,

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

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

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

by the rule *H-IND*,

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

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

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

Proof. Obvious from Proposition 1 and Definition 2. □

4.2. *Relating pseudorandomness and next-bit unpredictability*

The second example is the equivalence between pseudorandomness and next-bit unpredictability (Gol01). The notion of next-bit unpredictability can be reformulated in CSLR: a positive polynomial f such that $\vdash f : \Box\mathsf{Bits} \to \mathsf{Bits}$ is *next-bit unpredictable* if for all canonical numeral i such that $|i| < |f|$,

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

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,

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

Hence,

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

□

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 uses the hybrid technique. We define a hybrid function:

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

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

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

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

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

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. (MRST06) 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 (IK06). 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 $\texttt{let}\ i \leftarrow p(\boldsymbol{n})\ \texttt{in}\ e$, 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:

$$\texttt{let}\ i \leftarrow p(\boldsymbol{n})\ \texttt{in}\ e_1 \simeq \texttt{let}\ i \leftarrow p(\boldsymbol{n})\ \texttt{in}\ e_2,$$

according to the rule *SUB* we can only deduce

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

but never

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

However, the latter is used in many proofs in (IK06). 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 it might still involve reasoning about the probability. Meanwhile, our approach completely removes explicit reasoning about probability and complexity by using a type system based on safe recursion.

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 (Now08). It is in fact a formalization of the game-based

security proofs, an approach advocated by several researchers in cryptography (BR04; Sho04), 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 modeled directly as probabilistic distributions and the correctness of game transformations are verified in the proof-assistant, but the complexity-theoretic issues are not considered. A more sophisticated system is the CertiCrypt tool developed by Barthe et al. (BGZ09). Games are formalized as programs in an imperative language and transformations are proved using the relational Hoare logic. CertiCrypt is also implemented in Coq and is used to verify some interesting examples, e.g., the semantic security of OAEP. Another system designed by Backes et al. is based on a functional language with references and events and is implemented in Isabelle/HOL, but no cryptographic examples are given (BBU08).

Blanchet's CryptoVerif is another automated tool supporting game-based cryptographic proofs, but not based on any existing theorem provers (Bla06). 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. In particular, we do not need any explicit bound to impose the polynomial-time restraint, which allows us to remove completely computations of polynomials when reasoning about cryptographic constructions. This is the main advantage of using implicit complexity mechanism to build such a logic system.

Examples given in the paper are experimental and we are working on proving more realistic cryptographic constructions. Our recent work shows that game-based cryptographic proofs can be formalized and verified in the CSLR proof system, thanks to the general version of computational indistinguishability that we have defined in Definition 1. Furthermore, as higher-order functions are already native in the language, we expect that the system can be used to verify cryptographic protocols in the computational model.

Acknowledgement

I appreciate very much the inspiring communications with Jean Goubault-Larrecq. Also thank Gilles Barthe, Yuxin Deng, David Nowak, Aleksy Schubert and the TLCA reviewers for their discussions and feedbacks which help to improve the paper.

References

M. Backes, M. Berg, and D. Unruh. A formal language for cryptographic pseudocode. In *4th Workshop on Formal and Computational Cryptography (FCC 2008)*, 2008.

Stephen Bellantoni and Stephen A. Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.

G. Barthe, B. Grégoire, and S. Zanella. Formal certification of code-based cryptographic proofs. In *36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'2009)*, pages 90–101, 2009.

Bruno Blanchet. A computationally sound mechanized prover for security protocols. In *IEEE Symposium on Security and Privacy (S&P'06)*, pages 140–154, 2006.

M. Bellare and P. Rogaway. Code-based game-playing proofs and the security of triple encryption. Cryptology ePrint Archive, Report 2004/331, 2004.

Oded Goldreich. *The Foundations of Cryptography: Basic Tools*. Cambridge University Press, 2001.

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.

Martin Hofmann. Safe recursion with higher types and BCK-algebra. *Annals of Pure and Applied Logic*, 104(1-3):113–166, 2000.

Russell Impagliazzo and Bruce M. Kapron. Logics for reasoning about cryptographic constructions. *Journal of Computer and System Sciences*, 72(2):286–320, 2006.

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.

Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

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.

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.

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.

Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332, 2004.