Math. Struct. in Comp. Science (2010), *vol.* 20, *pp.* 951–975. © Cambridge University Press 2010
doi:10.1017/S0960129510000265

The computational SLR: a logic for reasoning about computational indistinguishability

YU ZHANG[†]

State Key Laboratory for Computer Science, Institute of Software, Chinese Academy of Sciences, P.O. Box 8718, Beijing 100190, China and State Key Laboratory for Novel Software Technology, Nanjing University
Email: yzhang@ios.ac.cn

Received 1 December 2009; revised 1 May 2010

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. In particular, we define rules that directly justify the computational indistinguishability between programs, and then prove that these rules are sound with respect to the set-theoretic semantics, and thus 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 a 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, and, in particular, is used to define many security criteria. However, proving computational indistinguishability in traditional cryptography is usually done using a paper-and-pencil, semi-formal method, which is often error-prone, and becomes unreliable when the cryptographic constructions are complex. The aim of this paper is to design a formal system that can help us verify cryptographic proofs; our ultimate goal is a full or partial automatation of the verification process.

Noting that computational indistinguishability can be viewed as a special notion of equivalence between programs, we make use of techniques from the theory of programming languages. However, this first requires an appropriate language for expressing cryptographic constructions and adversaries. In particular, we shall only consider 'feasible' adversaries; more 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 was drawn to Hofmann's SLR system

[†] This work was partly supported under the grant KFKT2008B16.

(Hofmann 1998; Hofmann 2000), which is a functional programming language that implements Bellantoni and Cook's safe recursion (Bellantoni and Cook 1992). A very nice property of SLR is its characterisation of polynomial-time computations through typing. The probabilistic extension of SLR was studied in Mitchell *et al.* (1998), 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, and thus coincide with the traditional definition of computational indistinguishability. Reasoning about cryptographic constructions in the proof system is purely syntactic, without explicit analysis of the probability of program output and the complexity bound of adversaries.

The rest of the paper is organised as follows. Section 2 introduces the computational SLR, which is 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. We give some cryptographic examples using the proof system in Section 4 to illustrate its usability in cryptography. In Section 5 we give a summary of related work, and, finally, present our conclusions in Section 6.

2. The computational SLR

We begin by defining a language for expressing cryptographic constructions and adversaries, as well as the computational indistinguishability between programs. Because of the complexity consideration, the language should offer a mechanism for capturing the class of probabilistic polynomial-time computations. Bellantoni and Cook have proposed an alternative recursion model to the Turing-machine model, which is called *safe recursion* and defines exactly the functions that are computable in polynomial time on a Turing-machine (Bellantoni and Cook 1992). 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 safe variables *only*; recursion must take place on normal variables and intermediate recursion results are never sent to safe variables. When higher-order functions are included, step functions are also required to be linear, that is, intermediate recursive results can only be used once in each step.

Hofmann later developed a functional language called SLR to implement safe recursion (Hofmann 1998; Hofmann 2000). 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 this is not required 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 computational indistinguishability in our language.

2.1. *The non-polymorphic SLR for bitstrings*

Types are defined by:

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

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

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

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:

(1) $\Box\tau \to \tau'$ are modal functions with no restriction on the use of arguments;
(2) $\tau \to \tau'$ are non-modal functions where arguments must be safe values;
(3) $\tau \multimap \tau'$ 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 one of:

(1) $\mathfrak{m} = (\text{modal}, \text{non-linear})$ for $\Box\tau \to \tau'$;
(2) $\mathfrak{n} = (\text{non-modal}, \text{non-linear})$ for $\tau \to \tau'$; or
(3) $\mathfrak{l} = (\text{non-modal}, \text{linear})$ for $\tau \multimap \tau'$.

The aspects are ordered: $\mathfrak{m} \leqslant \mathfrak{n} \leqslant \mathfrak{l}$. They are also used to tag variables in typing contexts.

The type system also inherits its 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, which allows $\mathsf{Bits} \to \tau <: \mathsf{Bits} \multimap \tau$, states that bitstrings can be duplicated without violating linearity.

The expressions of SLR are defined by the following grammar:

$e_1, e_2, \ldots$	$::= x$	atomic variables
	$\mid$ `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

Where:

— $\mathtt{B}_0$ 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 to the left-hand end of u. We will 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.

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

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

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

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

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

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

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

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

Fig. 2. Typing rules for SLR.

— $\mathtt{case}_\tau$ 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 for 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 $x_1 :^{a_1} \tau_1, \ldots, x_n :^{a_n} \tau_n$, where $a_1, \ldots, a_n$ are aspects of $\{\mathfrak{m}, \mathfrak{n}, \mathfrak{l}\}$. The typing rules are given in Figure 2.

2.2. *The computational SLR*

The probabilistic extension of SLR was studied by Mitchell *et al.* by adding a random bit oracle to simulate the oracle tape in probabilistic Turing machines (Mitchell *et al.* 1998). 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 onto the language. Hence we adopt a different syntax, which is taken from Moggi's computational λ-calculus (Moggi 1991), which is a pure

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

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

Fig. 3. Additional typing rules for computational SLR.

functional language where probabilistic computations are captured by monadic types. We call the language *computational SLR* and will often abbreviate it to *CSLR*.

Types in CSLR are extended with a unary type constructor

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

which is taken 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 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:

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

where:

— `rand` is a constant returning a random bit 0 or 1, each with probability 1/2.
— $\texttt{val}(e)$ is the trivial (deterministic) computation, which returns e with 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 will sometimes abbreviate programs of the form $\texttt{bind}\ x_1 = e_1\ \texttt{in}\ \ldots \texttt{bind}\ x_n = e_n\ \texttt{in}\ e$ by $\texttt{bind}\ (\,x_1 = e_1, \ldots, x_n = e_n\,)\ \texttt{in}\ e$. The order of some bindings must be carefully maintained 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:

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

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

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

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, respectively. To interpret the probabilistic computations, we adopt the probabilistic monad defined in Ramsey and Pfeffer (2002): if A is set, we write $\mathscr{D}_A : A \to [0, 1]$ for the set of probability mass functions over A. The original monad in Ramsey and Pfeffer (2002) 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 simplicity, in this paper we will work with mass functions instead of measures. Note that the monad is not the same as the one defined in Mitchell *et al.* (1998), which is used to keep track of the bits read from the oracle tape rather than for reasoning about probabilities.

When d is a mass function of $\mathscr{D}_A$ and $a \in A$, we also write $\mathbf{Pr}[a \leftarrow d]$ for the probability $d(a)$. If there are finitely many elements in $d \in \mathscr{D}_A$, 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 normal products and tensor products, or between the three sorts of function spaces.

A very nice property of SLR is a characterisation of polynomial-time computations (the class PTIME) through typing.

Theorem 1 (Hofmann (2000)). The set-theoretic interpretations of closed terms of type

$$\Box\mathsf{Bits} \to \mathsf{Bits}$$

in SLR define exactly the polynomial-time computable functions.

It is worth mentioning that the inclusion of tensor products and linear functions in SLR, which lead to a relatively complex type-checking that will probably not be an easy task for cryptographers, are not required for capturing PTIME computations unless we consider higher-order recursions, which can often make the programming in SLR easier. In practice, a simpler language without higher-order recursion (and consequently with no need for tensor products and linear functions) is probably enough for cryptographic use, but we will keep the language as it is to make it 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

Interpretation of types:

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

Interpretation of terms:

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

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

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 also be 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 enables us to reuse the result of Mitchell *et al.* (1998), when suitably adapted for CSLR.

Theorem 2 (Mitchell *et al.* 1998). The set-theoretic interpretations of closed terms of type $\Box\mathsf{Bits} \to \mathsf{TBits}$ in CSLR define exactly the 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, that is, $|u_1| = |u_2|$, we have $|p(u_1)| = |p(u_2)|$. When a term p is length sensitive, we write $|p|$ for the underlying length measure function, that is, $|p|(n) = |p(u)|$, where $|u| = n$. If p and q are two length-sensitive SLR-functions, we write $|p| < |q|$ to denote the fact that for any bitstring u, we have $|p(u)| < |q(u)|$, and similarly for $|p| > |q|$, $|p| = |q|$, and so on. A length-sensitive function is said to be *positive* if for every bitstring u, we have $|p(u)| > |u|$.

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, that is, $[\![p(u_1)]\!] = [\![p(u_2)]\!]$ if $|u_1| = |u_2|$. Note that we do not include 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 will often just refer to them as *polynomials*. A numerical polynomial is *canonical* if it returns the empty bitstring or bitstrings containing bit 1 only.

Intuitively, two probabilistic functions are computationally indistinguishable if the probability that any feasible adversary can distinguish between them becomes negligible when they take sufficiently large arguments. The following definition is adapted from the definition of computational indistinguishability in Goldreich (2001, Definition 3.2.2) for the setting of CSLR.

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

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

where ϵ denotes the empty bitstring.

In a cryptographic context, w is usually considered to be the security parameter.

Note that the above definition is more general than that of Goldreich (2001). Here we consider programs that can return values of an arbitrary type, while in the definition of Goldreich (2001), computational indistinguishability is only defined for distributions of bitstrings. One may argue that higher-order types are not necessary for reasoning about crypto-systems, but the general definition turned out to be very helpful when formalising 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 function $\boldsymbol{rs}$:

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

where h_{rs} is defined by

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

$\boldsymbol{rs}$ receives a bitstring and returns a uniformly random bitstring of the same length. It is easy to check that $\vdash h_{rs}\ : \Box\mathsf{Bits} \to \mathsf{TBits} \multimap \mathsf{TBits}$, and thus $\vdash \boldsymbol{rs}\ : \Box\mathsf{Bits} \to \mathsf{TBits}$.
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 function $\boldsymbol{conc}$:

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

where h_{conc} is defined by

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

h_{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 an SLR-term of type $\mathsf{Bits} \multimap \Box\mathsf{Bits} \to \mathsf{Bits}$, that is, 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 $\boldsymbol{conc}$ must be normal (modal). We often abbreviate $\boldsymbol{conc}(u, v)$ to $u \bullet v$.

— The head function $\boldsymbol{hd}$:

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

The tail function $\boldsymbol{tl}$:

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

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

— The split function $\boldsymbol{split}$:

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

where

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

$\boldsymbol{split}(x, n)$ splits the bitstring x into two bitstrings, the first of which is of length $|n|$ if $|n| \leqslant |x|$ or x otherwise. It is easy to check that $\boldsymbol{split}$ is SLR-definable and SLR-typable with type $\mathsf{Bits} \multimap \Box\mathsf{Bits} \to \mathsf{Bits} \otimes \mathsf{Bits}$. We can use $\boldsymbol{split}$ to define the prefix and suffix functions:

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

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

— The cut function $\boldsymbol{cut}$:

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

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

3. The proof system

In this section we present an equational proof system on top of CSLR, which we will use to justify computational indistinguishability between CSLR programs at the syntactic level.

The proof system contains two sets of rules:

(1) Rules for justifying semantic equivalence between CSLR programs (we write $e_1 \equiv e_2$ if e_1 and e_2 are semantically equivalent) – see Figure 5.
(2) Rules for justifying computational indistinguishability – see Figure 6.

The first set are standard rules for typed λ-calculi, with axioms (monad laws) for probabilistic computations. The rules in the second set are similar to those used in the logic of Impagliazzo and Kapron (2006), which we shall refer to as the IK-logic in the rest of the paper – they also define an equational proof system for computational indistinguishability based on their own arithmetic model. However, we do not have the *EDIT* rule for managing bitstrings that appears internally in their logic since CSLR has no primitive operations for editing bitstrings apart from the two bit constructors $\mathtt{B}_0, \mathtt{B}_1$. Many bitstring operations are defined as CSLR functions, and we will introduce a series of lemmas (see Section 3.2) that can be used in proofs in the same way as system rules. In this way we avoid the complexity analysis for 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 neighbouring 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 that is smaller than a polynomial, but the underlying support from the hybrid technique remains there.

Note that the rule *TRANS-INDIST* is safe and will not break the complexity constraint, because the number of times we apply a rule in a proof is irrelevant to the security parameter of the programs being tested.

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 of the 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.

Axioms:

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

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

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

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

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

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

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

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

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

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

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

Inference rules:

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

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

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

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

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

Fig. 5. Rules for semantic equivalence.

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

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

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

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

Fig. 6. Rules for computational indistinguishability.

Proof. We will prove that the rules in Figure 6 are sound. The soundness of the rule *EQUIV* is obvious.

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

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

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

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

Without loss of generality, we can assume that $n \geqslant n'$, so for every bitstring w such that $|w| \geqslant n$, we have

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

Since p is arbitrary, from Definition 1, we have $e_1 \simeq e_3$.

To prove the soundness of the rule *SUB*, we assume that there exists an adversary that can computationally distinguish the two terms in the conclusion part, and show that one can also build another adversary that 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

$|w| \geqslant n$ and

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

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 $\mathscr{A}'$ by

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

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

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

hence

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

which contradicts the premise $e_1 \simeq e_2$.

The soundness of the rule *H-IND* can be proved in a similar way to that of TRANS-INDIST. Let $\mathscr{A}$ be an arbitrary well-typed adversary and q be an arbitrary positive polynomial. Define another polynomial by $q' \stackrel{\text{def}}{=} \lambda x\,.\,\texttt{rec}(\texttt{nil}, \lambda m.\lambda r.q'(x) \bullet r, p(x))$. 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 any bitstring w whose length is larger than m,

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

Therefore,

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

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

3.2. *Useful lemmas for proving cryptographic constructions*

In this section we introduce some useful lemmas that will be used frequently in reasoning about cryptographic constructions. Most of the lemmas are about 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 number of the proofs can be done in the CSLR proof system. In this paper we only give a few proofs as examples – the others are left as exercises.

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 will only give the proof for the function $\boldsymbol{split}(u)$ – the proofs for all the others are similar.

We need to prove that, for all bitstrings n, m such that $|n| = |m|$, we have $[\![\boldsymbol{split}(u, n)]\!] = [\![\boldsymbol{split}(u, m)]\!]$, or that $\boldsymbol{split}(u, n) \equiv \boldsymbol{split}(u, m)$ according to Theorem 3. The proof is by induction on the length of the argument n.

The case where $|n| = 0$ is clear.

When $|n| > 0$, we suppose $n \equiv \mathtt{B}n'$ and $m \equiv \mathtt{B}m'$. Then

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

□

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

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

Proof. It is easy to deduce both results 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 $|u'| \leqslant |u|$, then $|u_1| = |u'|$.

Proof. We use induction on u'.

It is obvious that the lemma holds when $u' = \mathtt{nil}$.

Consider the induction step:

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

By the induction hypothesis, $|u_2| = |u| - |u_1| = |u| - |u'|$. If $|u'| = |u|$, then $|u_2| = 0$, that is, $u_2 \equiv \mathtt{nil}$ and $|u_1| = |u|$. So

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

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

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

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 $|\mathtt{B}u'| \leqslant |u|$. □

Lemma 4 (SPLIT-2). For all bitstrings u and u' such that $|u'| \geqslant |u|$,

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

Proof. First, for every bitstring u,

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

Now

$$\begin{aligned}
\textbf{\textit{split}}\,(u, \mathtt{B}_0 u') &\equiv \mathtt{rec}(\mathtt{nil} \otimes u, h_{\textbf{\textit{split}}}, \mathtt{B}_0 u') \\
&\equiv \mathtt{let}\ v_1 \otimes v_2 = \textbf{\textit{split}}\,(u, u')\ \mathtt{in} \\
&\quad\ \mathtt{case}(v_2, v_1 \otimes v_2, \lambda y.(v_1 \bullet 0) \otimes y, \lambda y.(v_2 \bullet 1) \otimes y) \\
&\equiv \mathtt{rec}(\mathtt{nil} \otimes u, h_{\textbf{\textit{split}}}, \mathtt{B}_1 u') \\
&\equiv \textbf{\textit{split}}\,(u, \mathtt{B}_1 u').
\end{aligned}$$

So for all bitstrings u_1 and u_2 such that $|u_1| = |u_2|$, we have $\textbf{\textit{split}}\,(u, u_1) \equiv \textbf{\textit{split}}\,(u, u_2)$.

For all bitstrings u and u' such that $|u'| = |u|$, we have $\textbf{\textit{split}}\,(u, u') \equiv u_1 \otimes u_2$ and $|u_1| = |u'|$ by Lemma 3, so $|u_2| = |u| - |u_1| = 0$, hence $u_2 \equiv \mathtt{nil}$, that is, $\textbf{\textit{split}}\,(u, u') \equiv u_1 \otimes \mathtt{nil}$. □

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

$$\begin{aligned}
\textbf{\textit{pref}}\,(u, \mathtt{nil}) &\equiv \mathtt{nil} \\
\textbf{\textit{pref}}\,(u, u') &\equiv u.
\end{aligned}$$

The proof is left as an exercise.

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

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

The proof is similar to the proof of Corollary 1 and is left as an exercise.

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

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

The proof is left as an exercise.

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

The proof is left as an exercise.

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

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

Proof. We use induction on the length of u. When $|u| = 0$, that is, $u \equiv \texttt{nil}$, we have

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

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

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

Then

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

□

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

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

The proof is left as an exercise.

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

The proof is left as an exercise.

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

The proof is left as an exercise.

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

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

The proof is left as an exercise.

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

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

The proof is left as an exercise.

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

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

The proof is left as an exercise.

4. Cryptographic examples

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

4.1. *Pseudorandom generators*

The first example verifies the correctness of Goldreich and Micali's construction of a pseudorandom generator (Goldreich 2001). This example also appears in Impagliazzo and Kapron (2006), but their proof has a subtle flaw (see Section 5 for an explanation).

We first reformulate the standard definition of a pseudorandom generator (Goldreich 2001, Definition 3.3.1) in CSLR.

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

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

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

We recall the construction of Goldreich (2001) (as reformulated in CSLR). Suppose g_1 is a PRG with expansion factor $|g_1|(x) = x + 1$, that is,

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

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

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

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

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

where the function R' is defined by

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

It is also easy to check 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 for the function G.

Lemma 14. For every bitstring n,

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

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

— When $|n| = 0$, that is, $n = \mathtt{nil}$:

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

— When $n = \mathtt{B}n'$ for some bitstring n', that is, $|n| = |n'| + 1$:

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

We next show that for all bitstrings u and n, we have $G(u,\mathtt{B}n) \equiv B(u)\bullet G(R(u),n)$. This is also proved by induction on $|n|$:

— When $|n| = 0$, that is, $n = \mathtt{nil}$:

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

— 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

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

we have

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

So

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

We now prove that the two programs in the assertion are computationally indistinguishable:

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

□

We next prove that given a polynomial p, it is easy to use G to construct a PRG with expansion factor $|p|$, and the proof is done in the CSLR proof system.

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

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

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

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

First,

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

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

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

So

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

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

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

So, by the rule *H-IND*,

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

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

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

Proof. The result is obvious from Proposition 1 and Definition 2. □

4.2. *Relating pseudorandomness and next-bit unpredictability*

The second example shows the equivalence between pseudorandomness and next-bit unpredictability (Goldreich 2001). The notion of next-bit unpredictability can be reformulated in CSLR as follows: a positive polynomial f such that $\vdash f : \Box\mathsf{Bits} \to \mathsf{Bits}$ is *next-bit unpredictable* if for any canonical numeral i such that $|i| < |f|$,

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

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,

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

□

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 by

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

It is easy to prove that, for all bitstrings u, v such that $|v| = |f(u)|$, we have $H(u,v,\texttt{nil}) \equiv v$ and $H(u,v,f(u)) \equiv f(u)$, hence

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

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

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

$$\equiv \lambda x\,.\,\mathtt{bind}\;(u = \boldsymbol{rs}(x), v = \boldsymbol{rs}(1\bullet \boldsymbol{suff}(f(x), \mathtt{B}_1 i))\,)\;\mathtt{in}\;\mathtt{val}(\boldsymbol{pref}(f(u), i)\bullet v)$$

(by the rule *RS-CONCAT*)

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

(by the rule *RS-EQUIV* since $|1\bullet \boldsymbol{suff}(f(x), \mathtt{B}_1 i)| = |\boldsymbol{suff}(f(x), i)|$)

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

(by the rule *RS-SPLIT*)

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

Hence, by the rule *H-IND*,

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

In other words, 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 separately in the previous two lemmas. □

5. Related work

Many researchers in cryptography have realised 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. A number of proof systems similar to ours have been proposed in recent years.

The PPC (probabilistic polynomial-time process calculus) system designed by Mitchell *et al.* (Mitchell *et al.* 2006) 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 using a form of probabilistic bisimulation. Interestingly, they mention that the 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 modelling protocols, but no such examples are given in their paper.

Impagliazzo and Kapron have proposed two logic systems for reasoning about cryptographic constructions (Impagliazzo and Kapron 2006). 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, unfortunately, it is 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 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:

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

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 form is used in many proofs in Impagliazzo and Kapron (2006). Furthermore, they claim that by introducing rules directly justifying 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. On the other hand, our approach completely removes explicit reasoning about probability and complexity by using a type system based on safe recursion.

Neither of the proof systems in PPC and IK-logic have been automated. However, Nowak has proposed a framework for formal verification of cryptographic primitives and has implemented it in the Coq proof-assistant (Nowak 2008). It is in fact a formalisation of game-based security proofs. This approach, in which proofs are carried out by generating a sequence of games, and transformations between the games must then be proved to be computationally sound, has been advocated by several researchers in cryptography (Bellare and Rogaway 2004; Shoup 2004). In Nowak's formalisation, games are modelled directly as probabilistic distributions, and the correctness of the game transformations is verified in the proof-assistant. However, the complexity-theoretic issues are not considered. A more sophisticated system is the CertiCrypt tool developed by Barthe *et al.* (Barthe *et al.* 2009). In this, games are formalised as programs in an imperative language and transformations are proved using the relational Hoare logic. CertiCrypt is also implemented in Coq, and has been used to verify some interesting examples, such as, the semantic security of OAEP. Another system designed by Backes *et al.* is based on a functional language with references and events, and has been implemented in Isabelle/HOL, but they have not given any cryptographic examples (Backes *et al.* 2008).

Blanchet's CryptoVerif is another automated tool that supports game-based cryptographic proofs, and is not based on any existing theorem provers (Blanchet 2006). Unlike the previously mentioned work, CryptoVerif aims to generate the sequence of games based on a collection of predefined transformations instead of verifying the computational soundness of transformations defined by users.

6. Conclusion

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

Unlike the related work mentioned in the previous section, where a language is either defined from scratch or no precise language definition is given, our language is an extention of Hofmann's

SLR, which has very solid mathematical support based on Bellantoni and Cook's safe recursion together with a nice mechanism for the characterisation of polynomial-time computations. In particular, we do not need an explicit bound to impose the polynomial-time restraint, which allows us to remove the computation of polynomials completely when reasoning about cryptographic constructions. This is the main advantage of using an implicit complexity mechanism to build such a logic system.

The examples given in the paper are just experimental and illustrative, but we are working on proving more realistic cryptographic constructions. Our recent work (Nowak and Zhang 2010) has shown that game-based cryptographic proofs can be formalised and verified in the CSLR proof system thanks to the general version of computational indistinguishability given by Definition 1. Furthermore, as higher-order functions are already included in the language, we expect that the system can be used to verify cryptographic protocols in the computational model.

Acknowledgement

I have very much appreciated some inspiring communications with Jean Goubault-Larrecq. I would also like to thank Gilles Barthe, Yuxin Deng, David Nowak, Aleksy Schubert and the TLCA reviewers for their comments and feedback, which have helped to improve the paper.

References

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

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

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

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

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

Goldreich, O. (2001) *The Foundations of Cryptography: Basic Tools*, Cambridge University Press.

Hofmann, M. (1998) 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). *Springer-Verlag Lecture Notes in Computer Science* **1414** 275–294.

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

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

Mitchell, J. C., Mitchell, M. and Scedrov, A. (1998) A linguistic characterization of bounded oracle computation and probabilistic polynomial time. In: *39th Annual Symposium on Foundations of Computer Science (FOCS'98)* 725–733.

Mitchell, J. C., Ramanathan, A., Scedrov, A. and Teague, V. (2006) A probabilistic polynomial-time process calculus for the analysis of cryptographic protocols. *Theoretical Computer Science* **353** (1–3) 118–164.

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

Nowak, D. (2008) A framework for game-based security proofs. In: 9th International Conference of Information and Communications Security (ICICS 2007). *Springer-Verlag Lecture Notes in Computer Science* **4861** 319–333.

Nowak, D. and Zhang, Y. (2010) A calculus for game-based security proofs. In: Proceedings of 4th International Conference on Provable Security (ProvSec'2010). *Springer-Verlag Lecture Notes in Computer Science* **6402**.

Ramsey, N. and Pfeffer, A. (2002) Stochastic lambda calculus and monads of probability distributions. In: *29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02)* 154–165.

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