Formal security proofs with minimal fuss: Implicit computational complexity at work

David Nowak, Yu Zhang

▶ To cite this version:

David Nowak, Yu Zhang. Formal security proofs with minimal fuss: Implicit computational complexity at work. Information and Computation, 2015, 241, pp.96-113. 10.1016/j.ic.2014.10.008 . hal-01144726

HAL Id: hal-01144726
https://hal.science/hal-01144726

Submitted on 26 Aug 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Formal Security Proofs with Minimal Fuss: Implicit Computational Complexity at Work☆

David Nowak[a], Yu Zhang[b]

[a] JFLI, CNRS & The University of Tokyo, Japan
[b] State Key Laboratory for Computer Science, ISCAS, China

Abstract

We show how implicit computational complexity can be used in order to increase confidence in game-based security proofs in cryptography. For this purpose we extend CSLR, a probabilistic lambda-calculus with a type system that guarantees the existence of a probabilistic polynomial-time bound on computations. This allows us to define cryptographic constructions, feasible adversaries, security notions, computational assumptions, game transformations, and game-based security proofs in a unified framework. We also show that the standard practice of cryptographers, ignoring that polynomial-time Turing machines cannot generate all uniform distributions, is actually sound. We illustrate our calculus on cryptographic constructions for public-key encryption and pseudorandom bit generation.

Keywords:
lambda-calculus, safe recursion, probabilistic computation, cryptography
2010 MSC: 03B40, 03D15, 60E05, 94A60

1. Introduction

When a new cryptographic scheme is published, it often comes with a security proof. It is unfortunately common that such schemes are vulnerable even though they have been supposedly proved secure — attacks are often found a few months or a few years after publication of the so-called security proof. This presents a serious demand for formal, ideally automated verification of security proofs in cryptography, which is well-known and acknowledged by cryptographers [19].

It is good practice to structure security proofs as a sequence of game transformations since it makes easier their checking by a third party [7, 34]. In this game-based approach, a security property is modeled as a probabilistic program

☆Preliminary results appeared in the proceedings of the 4th International Conference on Provable Security (ProvSec 2010) [32].

Preprint submitted to Elsevier *October 16, 2014*

implementing a game to be solved by the adversary. The adversary itself is modeled as an external probabilistic procedure interfaced with the game. Proving security amounts to proving that any adversary has at most a negligible advantage over an adversary playing against a perfectly secure scheme.

If however the adversary were given unlimited computational power, then it could break most cryptographic schemes without making them insecure in practice. It is thus necessary to restrict the computational power of the adversary so that the attacks are feasible. Cobham's thesis asserts that being feasible is the same as being computable in polynomial time [11]. Cryptographers follow Cobham's thesis in their security proofs by assuming that the adversary is computable in probabilistic polynomial time (for short, PPT), i.e., executable on a Turing machine extended with a read-only random tape that has been filled with random bits, and working in (worst-case) polynomial time. One way to take complexity into account in formal verification would be to formalize a precise execution model (e.g., Turing machines) and to explicitly count the number of steps necessary for the execution of the algorithm. Such approach would for the least be tedious and would give results depending on the particular execution model in use whereas one is mainly interested in the complexity class independent of execution models. Implicit computational complexity suggests a more convenient approach, which relates computations and/or programming languages with complexity classes without relying on specific execution models nor explicit counting of execution steps.

Although the languages of CertiCrypt [5], EasyCrypt [4] or CryptoVerif [8] can be used for writing game-based security proofs, a language that can automatically take into account complexity issues is still missing. In this paper we show how Computational SLR [36] (for short, CSLR), a probabilistic lambda-calculus with a type system that guarantees that computations are PPT, can be extended so as to allow for defining cryptographic constructions, feasible adversaries, security notions, computational assumptions, game transformations, and game-based security proofs in a unified framework.

In implementations of cryptographic schemes, because computers are based on binary digits, the cardinal of the support of a uniform distribution has to be a power of 2. Even at a theoretical level, probabilistic Turing machines used in the definition of PPT choose random numbers only among sets of cardinal a power of 2 [17]. In the case of another cardinal, the uniform distribution can only either be approximated or rely on code that is not guaranteed to terminate, although it will terminate with a probability arbitrarily close to 1 [23]. With arbitrary random choices that are used in games, one can define more distributions than those allowed by the definition of PPT. This raises a fundamental concern that is usually overlooked by cryptographers.

Related work. Over the last years, apart from the ones already mentioned above, other frameworks for machine-checking security proofs in cryptography have been proposed [2, 3, 12, 13, 30, 31]. However, these frameworks either ignore complexity-theoretic issues or postulate the complexity of the involved functions. More recently, in [20], Heraud and Nowak present their formalization in the

proof assistant Coq of Bellantoni and Cook's characterization of the class of polytime functions [6]. They show how their formalization can be used to deal with the complexity bound on the adversary in Nowak's toolbox [30, 31], and extend CertiCrypt [5] so as to alleviate the need for postulating the complexity of functions.

Since Bellantoni and Cook's characterization, there have been other ones. For instance, a recent one is given in [25].

Mitchell et al. have proposed a process calculus with bounded replications and messages to guarantee that processes are computable in polynomial time [28]. Messages can be terms of OSLR — Hofmann's SLR [21] with a random oracle [27]. Their calculus aim at being general enough to deal with cryptographic protocols, whereas we aim at a simpler calculus able to deal with cryptographic constructions.

Impagliazzo and Kapron have proposed two logics for reasoning about cryptographic constructions [24]. The first one is based on a non-standard arithmetic model, which, they prove, captures probabilistic polynomial-time computations. The second one is built on top of the first one, with rules justifying computational indistinguishability.

The above approaches are limited to the verification of cryptographic algorithms, and cannot deal with their implementations. This issue has been tackled by Affeldt et al. in [1] where it is shown how game-based security proofs can be conducted directly on implementations in assembly language.

Another probabilistic variant of Hofmann's SLR is proposed in [14]. In contrast to OSLR [27] and CSLR [36], the bound on the reduction time is here proved by syntactical means and thus makes explicit a notion of evaluation in polynomial time. However, for our purpose, it is enough to know that an adversary could be executed in polynomial time: we do not need an explicit reduction. Therefore, as it is the case with SLR and OSLR, there is no operational semantics for CSLR.

Contributions. We demonstrate the usefulness of implicit computational complexity in cryptography by writing game-based security proofs with CSLR, a lambda calculus whose type system allows for restricting the computational power of the adversary so that its attacks are feasible. By completeness, any adversary can be defined as a CSLR term of a certain type. Our approach has the advantage that it can automatically prove (by type inference [21]) that a program is PPT.

We show that the standard practice of cryptographers, ignoring that polynomial-time Turing machines cannot generate all uniform distributions, is actually sound.

With respect to feasible adversaries, we define a notion of game indistinguishability. Although, it is not stronger than the notion of computational indistinguishability of [36], it is simpler to prove and well-suited for formalizing game-based security proofs. We indeed show that this notion allows to easily model security definitions and computational assumptions. Moreover we show that computational indistinguishability implies game indistinguishability,

so that we can reuse as it is the equational proof system of [36]. More precisely, the former allows for any arbitrary use of the compared program by the adversary, while the latter provides more control over the adversary as it is usual in game-based security definitions, thus making game indistinguishability adequate.

CSLR, as initially defined by Zhang [36], does not allow superpolynomial-time computations (i.e., computations that are not bounded above by any polynomial) nor arbitrary uniform choices. Although this restriction makes sense for the cryptographic constructions and the adversary, the game-based approach to cryptographic proofs does not preclude the possibility of introducing games that perform superpolynomial-time computations or that use arbitrary uniform distributions. They are just idealized constructions that are used to define security notions but are not meant to make their way into implementations. We thus extend CSLR into $\mathrm{CSLR}_{\pi}^{\$}$ that includes CSLR as a sublanguage and that allows for superpolynomial-time computations and arbitrary uniform choices.

We illustrate the usability of our approach by proving formally in our proof system that the public-key encryption scheme ElGamal [16] and the pseudorandom bit generator of Blum, Blum and Shub [9] (for short, BBS) are secure in the appropriate senses stated by cryptographers [18, 9] and formalized here.

Outline. We introduce CSLR in Section 2. In Section 3, we discuss the problem of approximating uniform sampling from sets of arbitrary size using just fair coin tosses, and study the relation with perfect uniform sampling. In Section 4, we add the possibility of introducing superpolynomial-time primitives for defining security notions and equip the calculus with a notion of game indistinguishability. In Section 5, we illustrate the use of our calculus by proving formally with it the semantic security the ElGamal encryption scheme and the unpredictability of the Blum-Blum-Shub pseudorandom bit generator. Finally, we conclude in Section 6.

2. Computational SLR

Bellantoni and Cook have proposed to replace the model of Turing machines by their *safe recursion* scheme which defines exactly functions that are computable in polynomial time on a Turing-machine [6]. 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 normal variables. When higher-order recursors 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. Thanks to those syntactic restrictions, exponential-time computations are avoided. This is an elegant approach in the sense that polynomial-time computation is characterized without explicitly counting the number of computation steps.

Hofmann later developed a functional language called *SLR* to implement safe recursion [21, 22]. It provides a complete characterization through typing

of the complexity class of probabilistic polynomial-time computations. 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. Moreover there is a type-inference algorithm that can automatically determine the type of any expression [21]. Mitchell et al. have extended SLR by adding a random bit oracle to simulate the oracle tape in probabilistic Turing-machines [27].

More recently, Zhang has introduced CSLR, a non-polymorphic version of SLR extended with probabilistic computations and a primitive notion of bitstrings [36]. His use of monadic types [29], allows for an explicit distinction in CSLR between probabilistic and purely deterministic functions. This distinction was not possible with the extension by Mitchell et al. [27].

2.1. The language CSLR

We recall below the definition of CSLR and its main properties.

Types. Types are defined by:

$$\tau, \tau', \ldots \ ::= \ \mathsf{Bits} \mid \tau \times \tau' \mid \Box\tau \to \tau' \mid \tau \to \tau' \mid \tau \multimap \tau' \mid \mathsf{T}\tau$$

Bits is the base type for bitstrings. The monadic types $\mathsf{T}\tau$ capture probabilistic computations that produce a result of type τ. All other types are from Hofmann's SLR [22]. $\tau \times \tau'$ are cartesian product types. There are three kinds of functions: $\Box\tau \to \tau'$ are types for modal functions with no restriction on the use of their argument; $\tau \to \tau'$ are types for non-modal functions where the argument must be a safe value; $\tau \multimap \tau'$ are types for linear functions where the argument can only be used once. Note that linear types are not necessary when we do not have higher-order recursors, which are themselves not necessary for characterizing PTIME computations but can ease and simplify the programming of certain functions (such as defining the Blum-Blum-Shub pseudorandom bit generator in Section 4.4).

CSLR also has a sub-typing relation $<:$ between types. In particular, the sub-typing relation between the three kinds of functions is: $\tau \multimap \tau' <: \tau \to \tau' <: \Box\tau \to \tau'$. We also have $\mathsf{Bits} \to \tau <: \mathsf{Bits} \multimap \tau$, stating that bitstrings can be duplicated without violating linearity. The subtyping relation is inherited from SLR, with an additional rule saying that the constructor T preserves subtyping [36].

Expressions. Expressions of CSLR are defined by the following grammar:

$$\begin{array}{rcl} e_1, e_2, \ldots & ::= & x \mid \mathtt{nil} \mid \mathtt{B}_0 \mid \mathtt{B}_1 \mid \mathtt{case}_\tau \mid \mathtt{rec}_\tau \mid \lambda x.e \mid e_1 e_2 \\ & \mid & \langle e_1, e_2 \rangle \mid \mathtt{proj}_1 e \mid \mathtt{proj}_2 e \mid \mathtt{rand} \\ & \mid & \mathtt{return}(e) \mid \mathtt{bind}\ x \leftarrow e_1\ \mathtt{in}\ e_2 \end{array}$$

$\mathtt{B}_0$ and $\mathtt{B}_1$ are two constants for constructing bitstrings: if u is a bitstring, $\mathtt{B}_0 u$ (respectively, $\mathtt{B}_1 u$) is the new bitstring with a bit 0 (respectively, 1) added at

the left end of u. $\texttt{case}_\tau$ is the constant for case distinction: $\texttt{case}_\tau(n, \langle e, f_0, f_1\rangle)$ 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 $f_1(n)$ if the first bit of n is 1. $\texttt{rec}_\tau$ is the constant for recursion on bitstrings: $\texttt{rec}_\tau(e, f, n)$ returns e if n is empty, and $f(n, \texttt{rec}_\tau(e, f, n'))$ otherwise, where n' is the part of the bitstring n with its first bit cut off. $\texttt{rand}$ returns a random bit 0 or 1, each with the probability $\frac{1}{2}$. $\texttt{return}(e)$ is the trivial computation which returns e with probability 1. We note that CSLR has no restriction on e in general — it can be a probabilistic computation too (of type $\mathsf{T}\tau$), in which case $\texttt{return}(e)$ will be of type $\mathsf{TT}\tau$. $\texttt{bind}\ x \leftarrow e_1\ \texttt{in}\ e_2$ is the sequential computation which first computes the probabilistic computation e_1, binds its result to the variable x, then computes e_2. All other expressions are from Hoffman's SLR [22].

To ease the reading of CSLR terms, we shall use some syntactic sugar and abbreviations in the rest of the paper:

- $\lambda_ \,.\, e$ represents $\lambda x \,.\, e$ when x does not occur as a free variable in e;
- $x \stackrel{\$}{\leftarrow} e_1;\ e_2$ represents the probabilistic sequential computation

  $$\texttt{bind}\ x \leftarrow e_1\ \texttt{in}\ e_2$$

  where e_1 is a distribution in which a value x is chosen at random;
- $x \leftarrow e_1;\ e_2$ represents the deterministic sequential (call-by-value) computation $(\lambda x \,.\, e_2)e_1$;
- $\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2$ represents a simple case distinction $\texttt{case}(e, \langle e_2, \lambda_.e_2, \lambda_.e_1\rangle)$, which tests the first bit of e: if it is 1 then e_1 is executed, otherwise e_2 is executed;
- when a program F is defined recursively by $\lambda n \,.\, \texttt{rec}_\tau(e_1, e_2, n)$, we often write the definition as:

  $$F \stackrel{\text{def}}{=} \lambda n \,.\, \texttt{if}\ n \stackrel{?}{=} \texttt{nil}\ \texttt{then}\ e_1\ \texttt{else}\ e_2(n, F(\textbf{\textit{tail}}(n))),$$

  where $\stackrel{?}{=}$ and ***tail*** are respectively the equality test between two bitstrings and the function that remove the left-most bit from a bitstring. These functions can be defined in CSLR [36].

Type system. Typing assertions of expressions are of the form $\Gamma \vdash t : \tau$, where Γ is a typing context that assigns types and aspects (inherited from Hofmann's system) to variables. Intuitively, an aspect specifies how the variable can be used in the program. For instance, a linear aspect forces that the variable can be used only once. A typing 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. The type system for CSLR can be found in [36].

Operational semantics. We can define a reduction system for the computational SLR, and prove that every closed term has a canonical form. In particular, the canonical form of type Bits is:

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

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

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

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.

The language deals with bitstrings, but in many discussions of cryptography, it will be more convenient to see them as integers. We write $\widehat{b}$ for the integer value of the bitstring b.

Denotational semantics. The denotational semantics of CSLR is defined based on a set-theoretic model [36]. We write $\mathbb{B}$ for the set of bitstrings. To interpret the probabilistic computations, we adopt the probabilistic monad defined in [33]: if A is a set, we write $\mathcal{D}_A : A \to [0, 1]$ for the set of probability mass functions over A. The original monad in [33] 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 structures, not just discrete probabilities. But for the sake of simplicity, in this paper as well as in [36] we work on mass functions instead of measures. Note that the monad is not the one defined in [27], 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 $\mathbf{Pr}[d \rightsquigarrow a]$ 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 $p_i = d(a_i)$. When we restrict ourselves to finite distributions, our monad becomes identical to the one used in [30, 31].

With this monad, every computation type $\mathsf{T}\tau$ in CSLR will be interpreted as $\mathcal{D}_{[\![\tau]\!]}$, where $[\![\tau]\!]$ is the interpretation of τ. Expressions are interpreted within an environment which maps every free variable to an element of the corresponding type. In particular, the two computational constructions are interpreted as:

$$\begin{aligned} [\![\mathtt{return}(e)]\!]_\rho &= \{([\![e]\!]_\rho, 1)\} \\ \left[\!\!\left[x \stackrel{\$}{\leftarrow} e_1;\ e_2\right]\!\!\right]_\rho &= \underline{\lambda} v \,.\, \textstyle\sum_{v' \in [\![\tau]\!]} [\![e_2]\!]_{\rho[x \mapsto v']}(v) \times [\![e_1]\!]_\rho(v') \end{aligned}$$

where τ is the type of x (or $\mathsf{T}\tau$ is the type of e_1). Interpretation of other types and expressions is given in [36].

The main property of CSLR [27, 36] is:

Theorem 1. *The set-theoretic interpretations of closed terms of type* $\Box\mathsf{Bits} \to \mathsf{TBits}$ *in CSLR are exactly the functions that can be computed by a probabilistic Turing machine in polynomial time.*

This theorem implies that CSLR is expressive enough to model an adversary and to implement cryptographic constructions, as they both are probabilistic polynomial-time functions. We remark that adversaries can return values of types other than Bits (e.g., tuples of bitstrings), but we can always define adversaries as a PPT function of type $\Box\mathsf{Bits} \to \mathsf{T}\mathsf{Bits}$ by adopting some encoding of different types of values into bitstrings, so the theorem still applies. The same is true in case of functions with multiple arguments: we can uncurrify them and then adopt some encoding so that the theorem still applies.

The development of our system is based on CSLR. We often need to state that some functions are *deterministic*, however it is not sufficient to say the such functions are of non-monadic type (types with no constructor T), e.g., $\Box\mathsf{Bits} \to \mathsf{Bits}$. Because the language is functional, it is easy to define functions of non-monadic functions that involve probabilistic computations, for instance, $\lambda x\,.\,((\lambda y\,.\,1)\mathtt{rand})$. To specify a deterministic function in our system, we explicitly state that it is definable (and/or typable) in *SLR*.

An example of PPT function. The random bitstring generation is defined as follows:

$$\boldsymbol{rs} \stackrel{\text{def}}{=} \lambda n\,.\,\mathtt{if}\ (n \stackrel{?}{=} \mathtt{nil})\ \mathtt{then}\ \mathtt{return}(\mathtt{nil})\\ \mathtt{else}\ b \xleftarrow{\$} \mathtt{rand};\ u \xleftarrow{\$} \boldsymbol{rs}(\boldsymbol{tail}(n));\ \mathtt{return}(b\bullet u)$$

where $\bullet$ denotes the concatenation operation of bitstrings, which can be programmed and typed in CSLR [36]. $\boldsymbol{rs}$ receives a bitstring and returns a uniformly random bitstring of the same length. It can be checked that $\vdash \boldsymbol{rs} : \Box\mathsf{Bits} \to \mathsf{T}\mathsf{Bits}$.

Computational indistinguishability. A notion of *computational indistinguishability* in cryptography has been defined in the CSLR system [36].

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 closed CSLR term $\mathcal{A}$ of type $\Box\mathsf{Bits} \to \tau \to \mathsf{T}\mathsf{Bits}$ and every positive polynomial P, there exists some $N \in \mathbb{N}$ such that for all bitstrings η with $|\eta| \geq N$

$$|\mathbf{Pr}[[\![\mathcal{A}(\eta, f_1(\eta))]\!] \rightsquigarrow 1] - \mathbf{Pr}[[\![\mathcal{A}(\eta, f_2(\eta))]\!] \rightsquigarrow 1]| < \frac{1}{P(|\eta|)}$$

This definition is a reformulation of Definition 3.2.2 of [17] in CSLR. In particular, a CSLR term of type $\Box\mathsf{Bits} \to \mathsf{T}\tau$ defines a so-called probabilistic *ensemble*.

An equational proof system that can demonstrate computational indistinguishability between CSLR programs is also defined in [36]. The proof system consists of rules justifying semantic equivalence, and rules justifying computational indistinguishability. Program equivalence is written as $e_1 \equiv e_2$, indicating that e_1 and e_2 have the same denotation. In Figure 1, we list the proof rules that will be used in this paper. Please refer to [36] for the full proof system.

Rules justifying semantic equivalence:

$$\frac{}{x \xleftarrow{\$} \mathtt{return}(e_1);\ e_2 \equiv e_2[e_1/x]}\ \textit{AX-BIND-1} \qquad \frac{}{x \xleftarrow{\$} e;\ \mathtt{return}(x) \equiv e}\ \textit{AX-BIND-2}$$

$$\frac{}{x \xleftarrow{\$} (y \xleftarrow{\$} e_1;\ e_2);\ e_3 \equiv y \xleftarrow{\$} e_1;\ x \xleftarrow{\$} e_2;\ e_3}\ \textit{AX-BIND-3}$$

$$\frac{e \equiv e'}{\mathtt{return}(e) \equiv \mathtt{return}(e')}\ \textit{VAL} \qquad \frac{e_1 \equiv e_1' \quad e_2 \equiv e_2'}{x \xleftarrow{\$} e_1;\ e_2 \equiv x \xleftarrow{\$} e_1';\ e_2'}\ \textit{BIND}$$

Rules justifying computational indistinguishability:

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

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

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

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

Figure 1: Some rules from the CSLR proof system

3. Uniform sampling

As it is the case for computers, CSLR has binary digits as its fundamental representation of data. While uniform distributions are ubiquitous in cryptography, binary representation does not support an exact implementation of an arbitrary uniform distribution, particularly when the cardinal of its support is not a power of 2. This section discusses the approximation of uniform sampling in the setting of CSLR, and introduces an extension $\text{CSLR}^{\$}$ that includes a uniform sampling primitive.

3.1. Modeling uniform sampling in CSLR

In modern computers based on binary digits, implementing uniform distributions requires that the cardinal of the support of a uniform distribution should be a power of 2. In case of a different cardinal, such a distribution can be approximated by repeatedly selecting a random value in a larger distribution whose cardinal is a power of 2, until one obtain a value in the desired range or reach the maximal number of allowed attempts (*timeout*, which determines the precision of the approximation). In the latter case a default value is returned.[1]

[1] Another possible approach is to select a random value in a very large distribution, and then take the remainder modulo the cardinal of the desired distribution.

We implement this pseudo-uniform sampling in CSLR as follows:

$$\boldsymbol{zrand} \stackrel{\text{def}}{=} \lambda n\,.\,\lambda t\,.\,\texttt{if}\ t \stackrel{?}{=} \texttt{nil}\ \texttt{then}\ \texttt{return}(0^{|n|})$$
$$\texttt{else}\ v \stackrel{\$}{\leftarrow} \boldsymbol{rs}(n);\ \texttt{if}\ v \geq n\ \texttt{then}\ \boldsymbol{zrand}(n, \boldsymbol{tail}(t))$$
$$\texttt{else}\ \texttt{return}(v)$$

The program takes two arguments: the sampling range (represented by the value $\widehat{n}$) and the timeout (represented by $|t|$). The test $\geq$ can be programmed in CSLR. The timeout is represented by the length of the bitstring t for the sake of simplicity and readability of the program, but an alternative representation of using $\widehat{t}$ as the timeout is certainly acceptable.

The program ***zrand*** uses $u = 2^{\lceil \log_2 \widehat{n} \rceil}$ as the cardinal of the larger distribution and makes samplings in this distribution. The probability that one sampling falls outside the desired range is $\frac{u-\widehat{n}}{u}$, thus probability that $|t|$ consecutive attempts fail is $\left(\frac{u-\widehat{n}}{u}\right)^{|t|}$. ***zrand*** will return $0^{|n|}$ as the default value after $|t|$ consecutive failures, so the probability that a value smaller than $\widehat{n}$ but other than $0^{|n|}$ is returned is $\frac{1-\left(\frac{u-\widehat{n}}{u}\right)^{|t|}}{\widehat{n}}$, and the probability that $0^{|n|}$ is returned is $\frac{1+(\widehat{n}-1)\cdot\left(\frac{u-\widehat{n}}{u}\right)^{|t|}}{\widehat{n}}$.

Similarly, a finite group can be encoded in CSLR and multiplication and group exponentiation can be programmed (as implied by Theorem 1). In the sequel, we shall write $\mathbb{Z}_q$ (q a bitstring) for the set of bitstrings (of the same length than q) of $\{0, 1, \ldots, \widehat{q} - 1\}$, and $\mathbb{Z}_q^{\$}$ for the truly uniform distribution from $\mathbb{Z}_q$.

3.2. CSLR$^{\$}$

CSLR$^{\$}$ extends CSLR with a uniform sampling primitive `sample` of type $\mathsf{Bits} \multimap \mathsf{TBits}$. `sample` receives a bitstring as argument and returns uniformly a random bitstring of the same length whose integer value is strictly smaller than that of the argument, i.e., $[\![\texttt{sample}(q)]\!] = \mathbb{Z}_q^{\$}$ for every bitstring q. For instance, the distribution produced by $\texttt{sample}(101)$ is

$$[\![\texttt{sample}(101)]\!] = \{(000, \frac{1}{5}), \ldots, (100, \frac{1}{5})\}.$$

We can program a sampling from an arbitrary finite set (of CSLR definable elements, usually just bitstrings in cryptography) using `sample`, assuming that there is an index function over the set.

The type system of CSLR$^{\$}$ is extended with only the proper rules for `sample` and constants. Note that the type of `sample` is $\mathsf{Bits} \multimap \mathsf{TBits}$ so that it can accept arguments that are defined using linear resources.

The following lemma justifies the use of ***zrand*** to approximate the uniform sampling from $\mathbb{Z}_q$:

Lemma 1. *Let q be a closed CSLR term of type $\Box\mathsf{Bits} \to \mathsf{Bits}$. The probabilistic ensembles $[\![\lambda\eta\,.\,\boldsymbol{zrand}(q(\eta), \eta)]\!]$ and $[\![\lambda\eta\,.\,\texttt{sample}(q(\eta))]\!]$ are computationally indistinguishable, i.e., for every closed CSLR term $\mathcal{A}$ of type $\Box\mathsf{Bits} \to \tau \to \mathsf{TBits}$*

and every positive polynomial P, there exists some $N \in \mathbb{N}$ such that for all bitstrings η with $|\eta| \geq N$

$$|\mathbf{Pr}[[\![\mathcal{A}(\eta, \boldsymbol{zrand}(q(\eta), \eta))]\!] \rightsquigarrow 1] - \mathbf{Pr}[[\![\mathcal{A}(\eta, \mathtt{sample}(q(\eta)))]\!] \rightsquigarrow 1]| < \frac{1}{P(|\eta|)}.$$

Proof. We show that the two ensembles are *statistically close*:

$$\begin{aligned}
& \frac{1}{2} \cdot \Sigma_{v \in \mathbb{Z}_{\widehat{q(\eta)}}} \left|\mathbf{Pr}[[\![\boldsymbol{zrand}(q(\eta), \eta)]\!] \rightsquigarrow v] - \mathbf{Pr}[[\![\mathtt{sample}(q(\eta))]\!] \rightsquigarrow v]\right| \\
= & \frac{1}{2} \cdot \Sigma_{v \in \mathbb{Z}_{\widehat{q(\eta)}}} \left|\mathbf{Pr}[[\![\boldsymbol{zrand}(q(\eta), \eta)]\!] \rightsquigarrow v] - \mathbf{Pr}[\mathbb{Z}^{\$}_{\widehat{q(\eta)}} \rightsquigarrow v]\right| \\
= & \frac{1}{2} \cdot \left(\left| \frac{1 + (\widehat{q(\eta)} - 1) \cdot \varepsilon}{\widehat{q(\eta)}} - \frac{1}{\widehat{q(\eta)}} \right| + (\widehat{q(\eta)} - 1) \cdot \left| \frac{1 - \varepsilon}{\widehat{q(\eta)}} - \frac{1}{\widehat{q(\eta)}} \right| \right) \\
= & \frac{\widehat{q(\eta)} - 1}{\widehat{q(\eta)}} \cdot \varepsilon
\end{aligned}$$

is negligible with respect to $|\eta|$, where

$$\varepsilon = \left(\frac{2^{\lceil \log_2 \widehat{q(\eta)} \rceil} - \widehat{q(\eta)}}{2^{\lceil \log_2 \widehat{q(\eta)} \rceil}} \right)^{|\eta|} \leq (1/2)^{|\eta|}.$$

We can then conclude because statistical closeness implies computational indistinguishability (cf. Section 3.2.2 of [17]). □

In Lemma 1, the security parameter η is used directly as the timeout of ***zrand***. A more general implementation would instantiate the timeout by a polynomial of $|\eta|$, i.e., $\boldsymbol{zrand}(q(\eta), p(\eta))$ where p is a well-typed SLR function of type $\Box\mathsf{Bits} \to \mathsf{Bits}$. The choice of p will affect the final distribution of the program and consequently the advantage of adversaries in security experiments, but that remains negligible. It is possible to use CSLR to deal with exact security and the exact timeout with p is necessary in that case.

We say a CSLR program P with a free variable s of type $\Box\mathsf{Bits} \to \mathsf{TBits}$ is a *s-sampling-based program* if every occurence of the free variable s is in an application $s(t)$, with t an arbitrary CSLR term of type Bits. We call s the sampling function variable. The purpose of this definition is to be able to exclude higher-order arguments such as ***zrand*** or `sample` in the next proposition.

If a CSLR$^{\$}$ program P can be obtained from a s-sampling-based CSLR program P' by replacing all s with `sample`, i.e., $P = P'[\mathtt{sample}/s]$, we say that P is a *well-formed sampling program.*

Consider the computational indistinguishability as defined in Definition 1, but now programs (except for adversaries) can be defined in CSLR$^{\$}$, i.e., their definitions can include the primitive `sample`. Lemma 1 implies that we can freely replace the approximate uniform sampling ***zrand*** by the truly uniform sampling `sample` or vice versa in sampling-based CSLR programs, without affecting the computational indistinguishability.

Proposition 1. *For any arbitrary s-sampling-based CSLR program P of type $\Box\mathsf{Bits} \to \tau$, with the sampling function variable s as the only free variable,*

$$\lambda\eta \,.\, P(\eta)[\lambda x \,.\, \boldsymbol{zrand}(x,\eta)/s] \simeq \lambda\eta \,.\, P(\eta)[\mathtt{sample}/s].$$

Proof. First, it is easy to show that for n pairs of computationally indistinguishable CSLR programs $(f_1^1, f_2^1), \ldots, (f_1^n, f_2^n)$ (n an arbitrary number), it holds that $\lambda\eta \,.\, (f_1^1(\eta), \ldots, f_1^n(\eta)) \simeq \lambda\eta \,.\, (f_2^1(\eta), \ldots, f_2^n(\eta))$.

Next we define

- P' by renaming each occurence of s by a distinct fresh variables $s_1, \ldots, s_n$ in P so that we have $P'[s/s_1, \ldots, s/s_n] = P$;
- Q' by replacing each subterm $s_i(t_i)$ in P' by a distinct fresh variable v_i, i.e., $Q'[s_1(t_1)/v_1, \ldots, s_n(t_n)/v_n] = P'$ (it is clear that each t_i has either no free variable or only η as a free variable);
- Q by replacing each subterm v_i in Q' by a projection $\mathtt{proj}_i(v)$ with v a fresh variable, i.e., $Q = Q'[\mathtt{proj}_1(v)/v_1, \ldots, \mathtt{proj}_n(v)/v_n]$.

Clearly, Q is still a CSLR program and

$$\begin{aligned} P(\eta)[\mathtt{sample}/s] &\equiv Q(\eta)[(\mathtt{sample}(t_1), \ldots, \mathtt{sample}(t_n))/v] \\ P(\eta)[\lambda x \,.\, \boldsymbol{zrand}(x,\eta)/s] &\equiv Q(\eta)[(\boldsymbol{zrand}(t_1,\eta), \ldots, \boldsymbol{zrand}(t_n,\eta))/v] \end{aligned}$$

where $\equiv$ denotes program equivalence as defined in the proof system of CSLR in [36]. For an arbitrary CSLR adversary $\mathcal{A}$,

$$\begin{aligned} \mathcal{A}(\eta, P(\eta)[\mathtt{sample}/s]) &\equiv \mathcal{A}(\eta, Q(\eta)[(\mathtt{sample}(t_1), \ldots, \mathtt{sample}(t_n))/v] \\ &\equiv \mathcal{A}'(\eta, (\mathtt{sample}(t_1), \ldots, \mathtt{sample}(t_n))) \\ \mathcal{A}(\eta, P(\eta)[\lambda x \,.\, \boldsymbol{zrand}(x,\eta)/s]) &\equiv \mathcal{A}(\eta, Q(\eta)[(\boldsymbol{zrand}(t_1), \ldots, \boldsymbol{zrand}(t_n))/v] \\ &\equiv \mathcal{A}'(\eta, (\boldsymbol{zrand}(t_1), \ldots, \boldsymbol{zrand}(t_n))) \end{aligned}$$

where $\mathcal{A}' = \lambda\eta \,.\, \lambda f \,.\, \mathcal{A}(\eta, Q(\eta)[f(\eta)/v])$. Because $\lambda\eta \,.\, \boldsymbol{zrand}(t_i,\eta) \simeq \lambda\eta \,.\, \mathtt{sample}(t_i)$ ($i = 1, \ldots n$) by Lemma 1,

$$\lambda\eta \,.\, (\boldsymbol{zrand}(t_1,\eta), \ldots, \boldsymbol{zrand}(t_n,\eta)) \simeq \lambda\eta \,.\, (\mathtt{sample}(t_i), \ldots, \mathtt{sample}(t_n)),$$

therefore

$$\begin{aligned} & |\mathbf{Pr}[[\![\mathcal{A}(\eta, P(\eta)[\lambda x \,.\, \boldsymbol{zrand}(x,\eta)/s])]\!] \rightsquigarrow 1] - \mathbf{Pr}[[\![\mathcal{A}(\eta, P(\eta)[\mathtt{sample}/s])]\!] \rightsquigarrow 1]| \\ = \; & |\mathbf{Pr}[[\![\mathcal{A}'(\eta, (\boldsymbol{zrand}(t_1,\eta), \ldots, \boldsymbol{zrand}(t_n,\eta)))]\!] \rightsquigarrow 1] \\ & -\mathbf{Pr}[[\![\mathcal{A}'(\eta, (\mathtt{sample}(t_1), \ldots, \mathtt{sample}(t_n)))]\!] \rightsquigarrow 1]| \end{aligned}$$

is negligible since $\mathcal{A}'$ is also a valid CSLR adversary. $\Box$

3.3. Uniform computational indistinguishability

The definition of computational indistinguishability (as in Definition 1 or its original form in cryptography, e.g., Definition 3.2.2 in [17]) enforces that adversaries must be in the complexity class PPT. In our setting, it means that an adversary must be definable in the original CSLR without using the primitive `sample`. However it is a standard practice of cryptographers to ignore that polynomial-time Turing machines cannot generate all distributions. We formally show in this section that this practice is actually sound. Indeed, we can allow adversaries to use uniform sampling primitive `sample` and such adversaries gain no significantly larger capability than PPT adversaries in terms of computational indistinguishability.

Let $\mathcal{A}$ be a closed CSLR$^\$$ program of type $\Box\mathsf{Bits} \to \tau \to \mathsf{TBits}$ with τ an arbitrary type. We call $\mathcal{A}$ a *uniform adversary* if every occurence of `sample` is in an application $\texttt{sample}(t)$ for some subterm t.

Definition 2 (Uniform comp. ind.). Two CSLR$^\$$ terms f_1 and f_2, both of type $\Box\mathsf{Bits} \to \tau$, are *uniform computationally indistinguishable* (written as $f_1 \simeq^\$ f_2$) if for every uniform adversary $\mathcal{A}$ of type $\Box\mathsf{Bits} \to \tau \to \mathsf{TBits}$ and every positive polynomial P, there exists some $N \in \mathbb{N}$ such that for all bitstrings η with $|\eta| \geq N$

$$|\mathbf{Pr}[[\![\mathcal{A}(\eta, f_1(\eta))]\!] \rightsquigarrow 1] - \mathbf{Pr}[[\![\mathcal{A}(\eta, f_2(\eta))]\!] \rightsquigarrow 1]| < \frac{1}{P(|\eta|)}.$$

For the sake of clarity, we shall sometimes refer to the original definition of computational indistinguishability where adversaries are not allowed to use `sample` as PPT-computational indistinguishability. The following proposition states that uniform computational indistinguishability is equivalent to PPT-computational indistinguishability.

Proposition 2. *For every pair of well-formed sampling CSLR$^\$$ programs P_1, P_2 of type $\Box\mathsf{Bits} \to \tau$, $P_1 \simeq P_2$ if and only if $P_1 \simeq^\$ P_2$.*

Proof. Clearly every PPT adversary is also a uniform adversary whose definition does not include `sample`, hence uniform computational indistinguishability implies PPT-computational indistinguishability.

In the proof, we write

$$\mathbf{Pr}[[\![Q_1]\!] \rightsquigarrow 1] \doteq \mathbf{Pr}[[\![Q_2]\!] \rightsquigarrow 1]$$

if $|\mathbf{Pr}[[\![Q_1]\!] \rightsquigarrow 1] - \mathbf{Pr}[[\![Q_2]\!] \rightsquigarrow 1]|$ is negligible w.r.t. parameter $|\eta|$, where η appears as a bitstring in Q_1 and Q_2.

For the reverse direction, with an arbitrary uniform adversary $\mathcal{A}$, we define

$$\begin{aligned} \mathcal{A}' &= \lambda\eta \,.\, \mathcal{A}[\lambda x \,.\, \boldsymbol{zrand}(x,\eta)/\texttt{sample}], \\ P'_i &= \lambda\eta \,.\, P_i(\eta)[\lambda x \,.\, \boldsymbol{zrand}(x,\eta)/\texttt{sample}]. \end{aligned}$$

Clearly, $\mathcal{A}'$ is a PPT adversary and we have

$$\mathcal{A}'(\eta, P'_i(\eta)) \equiv \mathcal{A}(\eta, P_i(\eta))[\lambda x \,.\, \boldsymbol{zrand}(x,\eta)/\texttt{sample}].$$

We can prove that

$$
\begin{array}{ll}
& \mathbf{Pr}[[\![\mathcal{A}(\eta, P_1(\eta))]\!] \rightsquigarrow 1] \\
\doteq & \mathbf{Pr}[[\![\mathcal{A}'(\eta, P_1'(\eta))]\!] \rightsquigarrow 1] \\
& \quad \text{(by Proposition 1, } \mathcal{A}(\eta, P_1(\eta)) \simeq \mathcal{A}'(\eta, P_1'(\eta))) \\
\doteq & \mathbf{Pr}[[\![\mathcal{A}'(\eta, P_1(\eta))]\!] \rightsquigarrow 1] \\
& \quad \text{(by Proposition 1, } P_1 \simeq P_1' \text{ and } \mathcal{A}' \text{ being the PPT adversary)} \\
\doteq & \mathbf{Pr}[[\![\mathcal{A}'(\eta, P_2(\eta))]\!] \rightsquigarrow 1] \\
& \quad \text{(by hypothesis } P_1 \simeq P_2 \text{ and } \mathcal{A}' \text{ being the PPT adversary)} \\
\doteq & \mathbf{Pr}[[\![\mathcal{A}'(\eta, P_2'(\eta))]\!] \rightsquigarrow 1] \\
& \quad \text{(by Proposition 1, } P_2 \simeq P_2' \text{ and } \mathcal{A}' \text{ being the PPT adversary)} \\
\doteq & \mathbf{Pr}[[\![\mathcal{A}(\eta, P_2(\eta))]\!] \rightsquigarrow 1] \\
& \quad \text{(by Proposition 1, } \mathcal{A}(\eta, P_2(\eta)) \simeq \mathcal{A}'(\eta, P_2'(\eta))),
\end{array}
$$

therefore $P_1 \simeq^{\$} P_2$. □

Proposition 2 suggests that we can replace the notion of computational indistinguishability with the more general notion of uniform computational indistinguishability and use the CSLR proof system as it is.

4. A game-based proof system

CSLR by itself only considers computations based on binary digits and does not allow superpolynomial-time computations. Section 3 shows that we can introduce uniform sampling into CSLR and the proof system for computational indistinguishability remains valid. The complexity restriction makes sense for the cryptographic constructions and the adversary, however the game-based approach to cryptographic proofs does not preclude the possibility of introducing games that perform superpolynomial-time computations — they are just idealized constructions that are used to define security notions but are not meant to make their way into implementations.

In this section, we extend CSLR both with superpolynomial-time computations and arbitrary uniform choices.

4.1. CSLR$^{\$}_{\pi}$

CSLR$^{\$}_{\pi}$ extends CSLR with the uniform sampling primitive `sample` and a set π of superpolynomial-time primitives.

The type system of CSLR$^{\$}_{\pi}$ is extended with only the proper rules for `sample` and superpolynomial-time constants in π. Note that in CSLR$^{\$}_{\pi}$ we do not care any more about the complexity class that can be characterized using the type system[2] — the language and type system of CSLR$^{\$}_{\pi}$ are there for defining and describing security notions, not adversaries.

[2]Nevertheless, one might expect that the complexity class characterized by CSLR$^{\$}_{\pi}$ is PPT^X, where X is the smallest complexity class in which additional constants can be defined, but the exact relation between CSLR$^{\$}_{\pi}$ and the complexity classes remains to be clarified — the addition of the primitive `sample` alone allows for defining more distributions than in PPT.

We adopt the notion of uniform computational indistinguishability in Section 3.3, except that in CSLR$^\$_\pi$ we are considering indistinguishability between CSLR$^\$_\pi$ programs. Adversaries remains uniform adversaries that are definable in CSLR$^\$$.

Definition 3 (Comp. ind. in CSLR$^\$_\pi$). Two CSLR$^\$_\pi$ terms f_1 and f_2, both of type $\Box\mathsf{Bits} \to \tau$, are *computationally indistinguishable* (written as $f_1 \simeq^\$_\pi f_2$) if for every uniform adversary $\mathcal{A}$ of type $\Box\mathsf{Bits} \to \tau \to \mathsf{TBits}$ that is definable in CSLR$^\$$, and every positive polynomial P, there exists some $N \in \mathbb{N}$ such that for all bitstrings η with $|\eta| \geq N$

$$|\mathbf{Pr}[[\![\mathcal{A}(\eta, f_1(\eta))]\!] \rightsquigarrow 1] - \mathbf{Pr}[[\![\mathcal{A}(\eta, f_2(\eta))]\!] \rightsquigarrow 1]| < \frac{1}{P(|\eta|)}.$$

We call a probability distribution $\mathcal{D}$ over bitstrings a *CSLR$^\$_\pi$ distribution*, if it can be realized by a CSLR$^\$_\pi$ program (normally with `sample`), i.e., a closed CSLR$^\$_\pi$ term e of type TBits such that $[\![e]\!] = \mathcal{D}$, and we write $\lceil\mathcal{D}\rceil$ for the CSLR$^\$_\pi$ program that realizes the distribution.

For every closed CSLR$^\$_\pi$ program e of type $\mathsf{T}\tau$, we write $\operatorname{samp}(e)$ for the sample space of $[\![e]\!]$, i.e., the set of all values of $[\![\tau]\!]$ which e returns with nonzero probability.

Lemma 2. *For every pair of uniform CSLR$^\$_\pi$ distributions $\mathcal{C}$ and $\mathcal{D}$, if f is a CSLR$^\$_\pi$ function of type $\Box\mathsf{Bits} \to \mathsf{Bits}$ such that $[\![f]\!]$ is a n-to-1 surjection from $\operatorname{samp}(\mathcal{C})$ to $\operatorname{samp}(\mathcal{D})$, then $x \xleftarrow{\$} \lceil\mathcal{C}\rceil;\ \mathtt{return}(f(x)) \equiv \lceil\mathcal{D}\rceil$.*

Proof. The distribution of the lefthand side is

$$\left[\!\!\left[x \xleftarrow{\$} \lceil\mathcal{C}\rceil;\ \mathtt{return}(f(x))\right]\!\!\right] = \{([\![f]\!](v),\ n \times \frac{1}{|\operatorname{samp}(\mathcal{C})|})\ \mid\ v \in \operatorname{samp}(\mathcal{C})\}.$$

Because $[\![f]\!]$ is a n-to-1 surjection from $\operatorname{samp}(\mathcal{C})$ to $\operatorname{samp}(\mathcal{D})$, we have

$$|\operatorname{samp}(\mathcal{C})| = n \cdot |\operatorname{samp}(\mathcal{D})|,$$

and it is clear that the above distribution equals $\mathcal{D}$. $\square$

4.2. Game indistinguishability

The notion of computational indistinguishability already allows us to perform some cryptographic proofs as shown in [36], but many cryptographers advocate the so-called game-based approach which structures security proofs as a sequence of game transformations [7, 34] and computational indistinguishability is not a practical notion for game-based proofs. CSLR$^\$_\pi$ can help to formalize game transformations and makes it feasible to automate the proof checking procedure.

In game-based proofs, an adversary involved in a game can be an arbitrary probabilistic polynomial-time program, hence it can be encoded as a well-formed CSLR$^\$$ sampling program of type $\Box\mathsf{Bits} \to \mathsf{T}\tau$, where the security parameter will

bound its running time, and τ is the type of messages returned by the adversary. A *game* is encoded as a closed, well-formed CSLR$^{\$}_{\pi}$ sampling function of type $\Box\mathsf{Bits} \to (\Box\mathsf{Bits} \to \mathsf{T}\tau) \to \mathsf{TBits}$ that takes the security parameter and the adversary as arguments and returns one bit denoting whether the adversary wins the game. We say two games are indistinguishable if no adversary can win one of the games with significantly larger probability than in the other.

Definition 4 (Game indistinguishability). Two CSLR$^{\$}_{\pi}$ games g_1 and g_2 are *game indistinguishable* (written as $g_1 \approx g_2$) if for every uniform CSLR$^{\$}$ adversary $\mathcal{A}$ of type $\Box\mathsf{Bits} \to \mathsf{T}\tau$, and every positive polynomial P, there exists some $N \in \mathbb{N}$ such that for all bitstrings η with $|\eta| \geq N$,

$$|\mathbf{Pr}[[\![g_1(\eta, \mathcal{A})]\!] \rightsquigarrow 1] - \mathbf{Pr}[[\![g_2(\eta, \mathcal{A})]\!] \rightsquigarrow 1]| < \frac{1}{P(|\eta|)}$$

The above definition formalizes the idea that the change between the two games g_1 and g_2 cannot be noticed by an adversary.

Intuitively, the difference between computational indistinguishability and game indistinguishability is that, the former allows for any arbitrary use of the compared program by the adversary, while the latter provides more control over the adversary as it is usual in game-based security definitions, thus making game indistinguishability adequate. Hence, game indistinguishability is no stronger than computational indistinguishability as proved in the following proposition. This is why we can sometimes use the CSLR$^{\$}_{\pi}$ proof system, which is designed for proving computational indistinguishability, for proving game indistinguishability.

Proposition 3. *Computational indistinguishability in CSLR$^{\$}_{\pi}$ implies game indistinguishability.*

Proof. Let g_1 and g_2 be two arbitrary games of type $\Box\mathsf{Bits} \to (\Box\mathsf{Bits} \to \mathsf{T}\tau) \to \mathsf{TBits}$. For every uniform CSLR$^{\$}$ adversary $\mathcal{A}$ of type $\Box\mathsf{Bits} \to \mathsf{T}\tau$, construct the following adversary $\mathcal{A}'$:

$$\begin{aligned} \lambda\eta\,.\,\lambda g\,.\;\; & b \xleftarrow{\$} g(\eta, \mathcal{A}); \\ & \mathtt{if}\ b \stackrel{?}{=} 1\ \mathtt{then}\ \mathtt{return}(1)\ \mathtt{else}\ \mathtt{return}(0), \end{aligned}$$

and it can be checked that $\mathcal{A}'$ is still a uniform CSLR$^{\$}$ adversary and $\mathbf{Pr}[[\![\mathcal{A}'(\eta, g_i(\eta))]\!] \rightsquigarrow 1] = \mathbf{Pr}[[\![g_i(\eta, \mathcal{A})]\!] \rightsquigarrow 1]$. Because g_1 and g_2 are computationally indistinguishable, $|\mathbf{Pr}[[\![\mathcal{A}'(\eta, g_1(\eta))]\!] \rightsquigarrow 1] - \mathbf{Pr}[[\![\mathcal{A}'(\eta, g_2(\eta))]\!] \rightsquigarrow 1]|$ is negligible. □

4.3. CSLR$^{\$}_{\pi}$ proof system

CSLR$^{\$}_{\pi}$ inherits most of the equational proof system of CSLR: All the rules for program equivalence in CSLR can be used directly in CSLR$^{\$}_{\pi}$. No extra rules are needed for the primitive `sample`, but we can add rules for constants in π if necessary. The four rules for proving computational indistinguishability

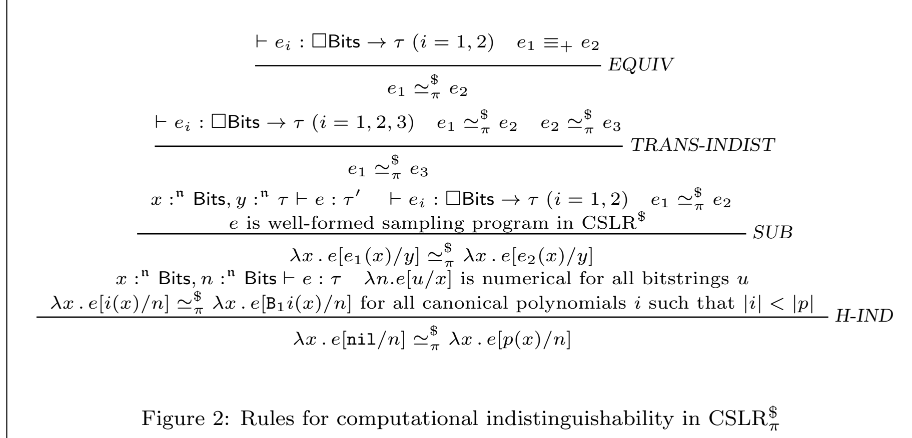

Figure 2: Rules for computational indistinguishability in CSLR$^{\$}_{\pi}$

remain the same as in CSLR (Figure 2) except that in the rule *SUB*, a new premise enforces that the substitution context (the term e) must be a well-formed sampling program in CSLR$^{\$}$, i.e., a program that uses `sample` properly and does not contain any superpolynomial-time constant.

The soundness of the system still holds and the proof just goes as for CSLR [36]. In particular, the proof for the rule *SUB* contains a construction of a new adversary with the context, which remains a uniform CSLR$^{\$}$ adversary thanks to the new premise.

We will also use the program equivalence defined in [36]. Roughly speaking, two terms e_1 and e_2 are equivalent (written $e_1 \equiv e_2$) if they have the same denotational semantics in any environment. Game transformation will consist in rewriting modulo the relation of game indistinguishability or computational indistinguishability or program equivalence. In particular, we will reuse as it is the equational proof system of [36] for game transformations.

Note that the rule *H-IND* is not used throughout this paper, but it is an important rule representing the hybrid proof technique that is frequently used in cryptography. Interested readers can find more detailed explanations and examples in [36].

Our further development in CSLR$^{\$}_{\pi}$ also relies on a few intermediate lemmas that are frequently used in game-based proofs. The first one states that an expression e which does not depend on a random bit b cannot guess this bit b.

Lemma 3. *If* $\Gamma \vdash e : \mathsf{TBits}$ *and, for all definable* $\rho \in [\![\Gamma]\!]$*, the domain of the distribution* $[\![e]\!]_{\rho}$ *is* $\{0, 1\}$*, then*

$$b \xleftarrow{\$} \mathtt{rand};\ x \xleftarrow{\$} e;\ \mathtt{return}(x \stackrel{?}{=} b) \equiv \mathtt{rand}$$

where $b \notin \mathrm{dom}(\Gamma)$.

Proof. We denote by e' the program on the left-hand side. For every definable

$\rho \in \Gamma$, $[\![e']\!]_\rho = \{(0, p_0), (1, p_1)\}$, where

$$p_0 \;=\; \mathbf{Pr}[[\![\mathtt{rand}]\!]_\rho \neq [\![e]\!]_\rho] = \frac{1}{2}\cdot\mathbf{Pr}[[\![e]\!]_\rho \neq 0] + \frac{1}{2}\cdot\mathbf{Pr}[[\![e]\!]_\rho \neq 1] = \frac{1}{2}$$
$$p_1 \;=\; \mathbf{Pr}[[\![\mathtt{rand}]\!]_\rho = [\![e]\!]_\rho] = \frac{1}{2}\cdot\mathbf{Pr}[[\![e]\!]_\rho = 0] + \frac{1}{2}\cdot\mathbf{Pr}[[\![e]\!]_\rho = 1] = \frac{1}{2}$$

hence $e' \equiv \mathtt{rand}$. □

The second lemma allows for a simplification when the semantics of a subexpression is a permutation. Remember that $\mathbb{Z}_q$ is the set of bitstrings defined at the end of Section 3.1.

Lemma 4. *Let f, f' be two closed CSLR$^\$_\pi$ terms of type $\square\mathsf{Bits} \to \mathsf{Bits}$ such that $[\![f]\!]$ is a permutation over $\mathbb{B}$, and, for every bitstring q, $[\![f']\!]$ is a permutation over $\{[\![f]\!](v) \mid v \in \mathbb{Z}_q\}$. It holds that*

$$\lambda\eta\,.\,x \stackrel{\$}{\leftarrow} \mathtt{sample}(q);\ \mathtt{return}(fx) \;\;\equiv\;\; \lambda\eta\,.\,x \stackrel{\$}{\leftarrow} \mathtt{sample}(q);\ \mathtt{return}(f'(fx))$$

Proof. Let e_1, e_2 denote the two programs on the left-hand and right-hand side respectively. Then for a given bitstring η, $[\![e_i(\eta)]\!]$ are two distributions over bitstrings, and $\mathrm{dom}([\![e_2(\eta)]\!]) = \{[\![f]\!](v) \mid v \in \mathbb{Z}_q\} = \mathrm{dom}([\![e_1(\eta)]\!])$ since $[\![f']\!]$ is a permutation over $\mathrm{dom}([\![e_1(\eta)]\!])$. Let e'_i be the program obtained from e_i by replacing $\mathtt{sample}(q)$ with a fresh variable w, i.e., $e'_i[\mathtt{sample}(q)/w] = e_i$, then $[\![e_i(\eta)]\!] = [\![\lambda w\,.\,e'_i(\eta)]\!](\mathbb{Z}^\$_q)$. By Lemma 3.1 of [31], $[\![e_1(\eta)]\!] = [\![\lambda w\,.\,e'_1(\eta)]\!](\mathbb{Z}^\$_q) = [\![\lambda w\,.\,e'_2(\eta)]\!](\mathbb{Z}^\$_q) = [\![e_2(\eta)]\!]$ as $[\![f']\!]$ is a permutation. □

4.4. Cryptographic constructions in CSLR$^\$_\pi$

This section presents examples of cryptographic constructions written in CSLR$^\$_\pi$. Note that they do not use any of the constants in π that are only to be used in the definitions of security notions.

The public-key encryption scheme ElGamal. Let G be a finite cyclic group of order q (depending on the security parameter η) and $\gamma \in G$ be a generator. The ElGamal encryption scheme [16] can be implemented in CSLR$^\$_\pi$ by the following programs:

- Key generation:

$$\boldsymbol{KG} \stackrel{\text{def}}{=} \lambda\eta\,.\,x \stackrel{\$}{\leftarrow} \mathtt{sample}(q);\ \mathtt{return}(\gamma^x, x)$$

  KG is of type $\square\mathsf{Bits} \to \mathsf{T}(\mathsf{Bits} \times \mathsf{Bits})$.

- Encryption:

$$\boldsymbol{Enc} \stackrel{\text{def}}{=} \lambda\eta\,.\,\lambda pk\,.\,\lambda m\,.\,y \stackrel{\$}{\leftarrow} \mathtt{sample}(q);\ \mathtt{return}(\gamma^y, pk^y * m)$$

  Enc is of type $\square\mathsf{Bits} \to \square\mathsf{Bits} \to \square\mathsf{Bits} \to \mathsf{T}(\mathsf{Bits} \times \mathsf{Bits})$.

- Decryption:

$$\textbf{\textit{Dec}} \stackrel{\text{def}}{=} \lambda\eta\,.\,\lambda sk\,.\,\lambda c\,.\,\texttt{proj}_2(c) * (\texttt{proj}_1(c)^{sk})^{-1}$$

  Dec is of type $\Box\mathsf{Bits} \to \Box\mathsf{Bits} \to \Box\mathsf{Bits} \to \mathsf{Bits}$, which does not involve monadic type because decryption is deterministic.

Note that when encoding cryptographic constructions in $\mathrm{CSLR}^{\$}_{\pi}$, we put the security parameter η explicitly as the argument of the programs. However, as we work on bitstrings in $\mathrm{CSLR}^{\$}_{\pi}$, the security parameter in traditional cryptographic contexts actually corresponds to $|\eta|$ here. In the case of ElGamal encryption, the group order q will be determined by η. Particularly, for the encryption scheme to be semantically secure, we must choose a suitable group such that the DDH assumption holds, and its order will be necessarily exponential w.r.t. $|\eta|$. There are efficient algorithms that compute a suitable DDH group given η, hence can be programmed in $\mathrm{CSLR}^{\$}_{\pi}$ without using any constant from π [10].

When `sample` is replaced by $\lambda x\,.\,\textbf{\textit{zrand}}(x,\eta)$ in the functions ***KG***, ***Enc***, ***Dec***, they are all computable in polynomial time w.r.t. η, even though ***Enc*** and ***Dec*** allow recursion on arguments other than η — these arguments (if valid) are all polynomially bound by η. Validity check of arguments are omitted in the implementation, but they can also be programmed in $\mathrm{CSLR}^{\$}_{\pi}$ without using any constant from π.

The Blum-Blum-Shub pseudorandom bit generator. The BBS generator defined in [9] is a deterministic function and can be programmed in $\mathrm{CSLR}^{\$}_{\pi}$ as follows:

$$\textbf{\textit{BBS}} \stackrel{\text{def}}{=} \lambda\eta\,.\,\lambda l\,.\,\lambda s\,.\,\textbf{\textit{bbsrec}}(\eta, l, s^2 \mathrm{mod}\ n)$$

where ***bbsrec*** is defined recursively as

$$\textbf{\textit{bbsrec}} \stackrel{\text{def}}{=} \lambda\eta\,.\,\lambda l\,.\,\lambda x\,.\,\texttt{if}\ l \stackrel{?}{=} \texttt{nil then nil else}\ \textbf{\textit{parity}}(x) \bullet \textbf{\textit{bbsrec}}(\eta, \textbf{\textit{tail}}(l), x^2 \mathrm{mod}\ n)$$

where n is determined by the security parameter η. ***BBS*** is a well-typed SLR-function of type $\Box\mathsf{Bits} \to \Box\mathsf{Bits} \to \Box\mathsf{Bits} \to \mathsf{Bits}$, with the second argument being the length of the resulted pseudorandom bitstring and the third argument being the seed, which is polynomially bound w.r.t. η.

4.5. Security notions in $CSLR^{\$}_{\pi}$

Security notions can be defined in term of game indistinguishability. We show how to use it to define some common security notions in cryptography.

Semantic security. A public-key encryption scheme $(\boldsymbol{KG}, \boldsymbol{Enc}, \boldsymbol{Dec})$ is said to be *semantically secure* [18] if:

$$\begin{array}{l} \lambda\eta\,.\,\lambda\mathcal{A}\,.\,(pk, sk) \stackrel{\$}{\leftarrow} \boldsymbol{KG}(\eta); \\ \qquad (m_0, m_1, \mathcal{A}') \stackrel{\$}{\leftarrow} \mathcal{A}(\eta, pk); \\ \qquad b \stackrel{\$}{\leftarrow} \texttt{rand}; \\ \qquad c \stackrel{\$}{\leftarrow} \boldsymbol{Enc}(\eta, m_b, pk); \\ \qquad b' \stackrel{\$}{\leftarrow} \mathcal{A}'(c); \\ \qquad \texttt{return}(b' \stackrel{?}{=} b) \end{array} \approx \lambda\eta\,.\,\lambda\mathcal{A}\,.\,\texttt{rand}$$

where $\mathcal{A}$ and $\mathcal{A}'$ are of types, respectively, $\Box\mathsf{Bits} \to \Box\tau_k \to \mathsf{T}(\tau_m \times \tau_m \times (\Box\tau_e \to \mathsf{T}\mathsf{Bits}))$ and $\Box\tau_e \to \mathsf{T}\mathsf{Bits}$. Note that τ_k, τ_e and τ_m are the respective types of public keys, cipher-texts and plain-texts, which can be tuples of bitstrings that are distinguished in the language. Roughly speaking, it means that any adversary $\mathcal{A}$ playing the semantic security game (left-side game) cannot do significantly better than a random player (right-side game). The semantic security game is to be read as follows: A pair (pk, sk) of public and secret keys is generated; the public key pk is passed to the adversary $\mathcal{A}$ which returns two messages m_1, m_2 and a function $\mathcal{A}'$, which can be seen as the continuation of the adversary $\mathcal{A}$ and contains necessary information that $\mathcal{A}$ has already obtained; one of the messages m_b, is selected at random and encrypted with the public key pk; the obtained cipher-text c is then passed to the function $\mathcal{A}'$, which returns its guess b' for the selected message; the result of the game indicates whether the adversary's guess is correct.

Left-bit unpredictability. An SLR-function F is *left-bit unpredictable* if:

$$\begin{array}{l} \lambda\eta\,.\,\lambda\mathcal{A}\,.\,s \stackrel{\$}{\leftarrow} \texttt{sample}(q);\ u \leftarrow F(\eta, s); \\ \qquad b \stackrel{\$}{\leftarrow} \mathcal{A}(\eta, \boldsymbol{tail}(u));\ \texttt{return}(b \stackrel{?}{=} \boldsymbol{head}(u)) \end{array} \approx \lambda\eta\,.\,\lambda\mathcal{A}\,.\,\texttt{rand} \qquad (1)$$

where $\mathcal{A}$ is of type $\Box\mathsf{Bits} \to \Box\mathsf{Bits} \to \mathsf{Bits}$. Roughly speaking, it means that any adversary $\mathcal{A}$ playing the unpredictability game (left-side game) cannot do significantly better than an adversary playing against a perfectly secure scheme (right-side game). The left-bit unpredictability game is to be read as follows: a seed s is selected at random in a set of cardinal q; the function F is then used to compute a pseudorandom bit sequence u of size $l(|q|) > |q|$ where l is a polynomial; the sequence u minus its first bit is passed to the adversary $\mathcal{A}$ which returns its guess b for the first bit; the result of the game indicates whether the adversary's answer is correct. It was proved by Yao in [35] that left-bit unpredictability is equivalent to passing all polynomial-time statistical tests.

A notion of next-bit unpredictability was defined in [36], but it is based on the sampling from bitstrings of a given length. We can generalize this notion and obtain another notion of left-bit unpredictability, which we shall refer to as *strong left-bit unpredictability* because it implies the game-based notion of left-bit unpredictability (1). An SLR-function F is *strongly left-bit unpredictable*

$$\lambda\eta\,.\,\lambda\mathcal{A}\,.\quad s \xleftarrow{\$} \mathtt{sample}(q);\ u \leftarrow F(\eta,s);\ b \xleftarrow{\$} \mathcal{A}(\eta, \boldsymbol{tail}(u));\ \mathtt{return}(b \stackrel{?}{=} \boldsymbol{head}(u))$$

$$\equiv\quad \lambda\eta\,.\,\lambda\mathcal{A}\,.\quad u \xleftarrow{\$} \begin{pmatrix} s \xleftarrow{\$} \mathtt{sample}(q); \\ \mathtt{return}(F(\eta,s)) \end{pmatrix};\ b \xleftarrow{\$} \mathcal{A}(\eta, \boldsymbol{tail}(u));\ \mathtt{return}(b \stackrel{?}{=} \boldsymbol{head}(u))$$

(By rules *AX-BIND-3* and *AX-BIND-1* of [36])

$$\approx\quad \lambda\eta\,.\,\lambda\mathcal{A}\,.\quad u \xleftarrow{\$} \begin{pmatrix} s \xleftarrow{\$} \mathtt{sample}(q); \\ b' \xleftarrow{\$} \mathtt{rand}; \\ \mathtt{return}(b' \bullet \boldsymbol{tail}(F(\eta,s))) \end{pmatrix};\ b \xleftarrow{\$} \mathcal{A}(\eta, \boldsymbol{tail}(u));\ \mathtt{return}(b \stackrel{?}{=} \boldsymbol{head}(u))$$

(By strong left-bit unpredictability)

$$\equiv\quad \lambda\eta\,.\,\lambda\mathcal{A}\,.\quad s \xleftarrow{\$} \mathtt{sample}(q);\ b' \xleftarrow{\$} \mathtt{rand};\ u \leftarrow b' \bullet \boldsymbol{tail}(F(\eta,s));\ b \xleftarrow{\$} \mathcal{A}(\eta, \boldsymbol{tail}(u));\ \mathtt{return}(b \stackrel{?}{=} \boldsymbol{head}(u))$$

(By rules *AX-BIND-3* and *AX-BIND-1* of [36])

$$\equiv\quad \lambda\eta\,.\,\lambda\mathcal{A}\,.\quad b' \xleftarrow{\$} \mathtt{rand};\ s \xleftarrow{\$} \mathtt{sample}(q);\ u \leftarrow b' \bullet \boldsymbol{tail}(F(\eta,s));\ b \xleftarrow{\$} \mathcal{A}(\eta, \boldsymbol{tail}(u));\ \mathtt{return}(b \stackrel{?}{=} b')$$

(By rules *AX-BIND-3* of [36])

$$\equiv\quad \lambda\eta\,.\,\lambda\mathcal{A}\,.\quad \mathtt{rand}$$

(By Lemma 3)

Figure 3: Proof of Proposition 4

if

$$\begin{array}{l}\lambda\eta\,.\,s \xleftarrow{\$} \mathtt{sample}(q); \\ \quad \mathtt{return}(F(\eta,s))\end{array} \quad \simeq^{\$}_{\pi} \quad \begin{array}{l}\lambda\eta\,.\,s \xleftarrow{\$} \mathtt{sample}(q); \\ \quad b \xleftarrow{\$} \mathtt{rand}; \\ \quad \mathtt{return}(b \bullet \boldsymbol{tail}(F(\eta,s)))\end{array}$$

Proposition 4. *Strong left-bit unpredictability implies left-bit unpredictability.*

Proof. The proof can be done using the $\mathrm{CSLR}^{\$}_{\pi}$ proof system. See Figure 3 for details. □

5. Applications

5.1. Semantic security of the ElGamal encryption scheme

In this section, we illustrate our proof system by proving the semantic security of the ElGamal encryption scheme. The security of ElGamal is based on the the Decisional Diffie-Hellman (DDH) assumption [15], which can also be defined in $\mathrm{CSLR}^{\$}_{\pi}$.

Decisional Diffie-Hellman assumption. Let q be a bitstring depending on the security parameter η, G be a finite cyclic group of order $\widehat{q}$ and $\gamma \in G$ be a generator. The Decisional Diffie-Hellman (DDH) assumption [15] states that, roughly speaking, no efficient algorithm can distinguish between triples of the form $(\gamma^x, \gamma^y, \gamma^{xy})$ and $(\gamma^x, \gamma^y, \gamma^z)$ where x, y and z are random numbers such

that $0 \leq x, y, z < \widehat{q}$.[3] It can be formalized in $\mathrm{CSLR}^{\$}_{\pi}$ as computational indistinguishability between two $\mathrm{CSLR}^{\$}_{\pi}$ programs: $\boldsymbol{DDHL} \simeq \boldsymbol{DDHR}$, where

$$\begin{aligned} \boldsymbol{DDHL} &= \lambda\eta\,.\, x \stackrel{\$}{\leftarrow} \mathtt{sample}(q);\ y \stackrel{\$}{\leftarrow} \mathtt{sample}(q); \\ &\quad \mathtt{return}(\gamma^x, \gamma^y, \gamma^{xy}) \\ \boldsymbol{DDHR} &= \lambda\eta\,.\, x \stackrel{\$}{\leftarrow} \mathtt{sample}(q);\ y \stackrel{\$}{\leftarrow} \mathtt{sample}(q);\ z \stackrel{\$}{\leftarrow} \mathtt{sample}(q); \\ &\quad \mathtt{return}(\gamma^x, \gamma^y, \gamma^z) \end{aligned}$$

Figure 4 shows the detailed proof of semantic security of the ElGamal encryption scheme, which follows the same structure as the one in [30], but here the type system of CSLR guarantees that the adversary is probabilistic polynomial-time. This was not dealt with in [30]. Moreover here all transformations are purely syntactic (thus allowing the immediate prospect of being implemented in an automated tool), while in [30] they were done at the semantics level.

Note that by using Lemma 4, we assume that the adversary $\mathcal{A}$ will not send any junk messages, i.e., bitstrings that are not elements of the group $\mathcal{G}_\eta$. This is considered as a trivial case in cryptography proofs because the ElGamal encryption procedure will automatically reject the junk messages. But in practice, in more complex crypto-systems, this may not be trivial at all. In our proof system, we can also consider the case where adversaries may send junk messages. It suffices to provide the corresponding code in the program ***Enc*** which tests the validity of incoming messages, and we can still prove semantic security in the $\mathrm{CSLR}^{\$}_{\pi}$ proof system. Another possibility would be to use a richer type system to reject adversaries returning junk.

We also note that by replacing all occurence of $\mathtt{sample}(q)$ in the proof with $\boldsymbol{zrand}(q, \eta)$, we immediately obtain a proof for the semantic security of an implementation of the ElGamal scheme. The validity of the new proof is justified by Propositions 1 and 2.

5.2. Unpredictability of the BBS pseudorandom bit generator

CLSR+ also allows us to formalize directly the proof of unpredictability given in [31] for the pseudorandom bit generator BBS. The proof requires a test for quadratic residuosity which is a superpolynomial-time computation — it can be introduced into $\mathrm{CSLR}^{\$}_{\pi}$ as a constant (in π). Moreover this proof is based on the Quadratic Residuosity Assumption stated below that uses arbitrary uniform choices.

Let n be a positive number and $\mathbb{Z}_n$ be the set of integers modulo n. The multiplicative group of $\mathbb{Z}_n$ is written $\mathbb{Z}_n^*$ and consists of the subset of integers modulo n which are coprime with n. An integer $x \in \mathbb{Z}_n^*$ is a quadratic residue modulo n iff there exists a $y \in \mathbb{Z}_n^*$ such that $y^2 = x \pmod n$. Such a y is called a square root of x modulo n. We write $\mathbb{Z}_n^*(+1)$ for the subset of integers in $\mathbb{Z}_n^*$ with Jacobi symbol equal to 1.

[3] We do not assume that $\widehat{q}$ is prime. However most groups in which DDH is believed to be true have prime order [10].

$$\lambda\eta\,.\,\lambda\mathcal{A}\,.\,\langle pk, sk\rangle \xleftarrow{\$} \boldsymbol{KG}(\eta);\ \langle m_0, m_1, \mathcal{A}'\rangle \xleftarrow{\$} \mathcal{A}(\eta, pk);\ b \xleftarrow{\$} \texttt{rand};\ c \xleftarrow{\$} \boldsymbol{Enc}(\eta, pk, m_b);\ b' \xleftarrow{\$} \mathcal{A}'(c);\ \texttt{return}(b \stackrel{?}{=} b')$$

$$\equiv\ \lambda\eta\,.\,\lambda\mathcal{A}\,.\,\langle pk, sk\rangle \xleftarrow{\$} \left(\begin{array}{l} x \xleftarrow{\$} \texttt{sample}(q); \\ \texttt{return}(\gamma^x, x) \end{array}\right);\ \langle m_0, m_1, \mathcal{A}'\rangle \xleftarrow{\$} \mathcal{A}(\eta, pk);\ b \xleftarrow{\$} \texttt{rand};\ c \xleftarrow{\$} \left(\begin{array}{l} y \xleftarrow{\$} \texttt{sample}(q); \\ \texttt{return}(\gamma^y, pk^y * m_b) \end{array}\right);\ b' \xleftarrow{\$} \mathcal{A}'(c);\ \texttt{return}(b \stackrel{?}{=} b')$$

(Inline of definition of ***KG*** and ***Enc***)

$$\equiv\ \lambda\eta\,.\,\lambda\mathcal{A}\,.\,x \xleftarrow{\$} \texttt{sample}(q);\ y \xleftarrow{\$} \texttt{sample}(q);\ b \xleftarrow{\$} \texttt{rand};\ \langle m_0, m_1, \mathcal{A}'\rangle \xleftarrow{\$} \mathcal{A}(\eta, \gamma^x);\ b' \xleftarrow{\$} \mathcal{A}'(\gamma^y, (\gamma^x)^y * m_b);\ \texttt{return}(b \stackrel{?}{=} b')$$

(By the equivalence rules *AX-BIND-3* and *AX-BIND-1* in [36])

$$\equiv\ \lambda\eta\,.\,\lambda\mathcal{A}\,.\,v \xleftarrow{\$} \boldsymbol{DDHL}(\eta);\ b \xleftarrow{\$} \texttt{rand};\ \langle m_0, m_1, \mathcal{A}'\rangle \xleftarrow{\$} \mathcal{A}(\eta, \texttt{proj}_1(v));\ b' \xleftarrow{\$} \mathcal{A}'(\texttt{proj}_2(v), \texttt{proj}_3(v) * m_b);\ \texttt{return}(b \stackrel{?}{=} b')$$

(Inline of ***DDHL***)

$$\approx\ \lambda\eta\,.\,\lambda\mathcal{A}\,.\,v \xleftarrow{\$} \boldsymbol{DDHR}(\eta);\ b \xleftarrow{\$} \texttt{rand};\ \langle m_0, m_1, \mathcal{A}'\rangle \xleftarrow{\$} \mathcal{A}(\eta, \texttt{proj}_1(v));\ b' \xleftarrow{\$} \mathcal{A}'(\texttt{proj}_2(v), \texttt{proj}_3(v) * m_b);\ \texttt{return}(b \stackrel{?}{=} b')$$

(By DDH assumption and *SUB*)

$$\equiv\ \lambda\eta\,.\,\lambda\mathcal{A}\,.\,x \xleftarrow{\$} \texttt{sample}(q);\ y \xleftarrow{\$} \texttt{sample}(q);\ z \xleftarrow{\$} \texttt{sample}(q);\ b \xleftarrow{\$} \texttt{rand};\ \langle m_0, m_1, \mathcal{A}'\rangle \xleftarrow{\$} \mathcal{A}(\eta, \gamma^x);\ b' \xleftarrow{\$} \mathcal{A}'(\gamma^y, \gamma^z * m_b);\ \texttt{return}(b \stackrel{?}{=} b')$$

(Inline of ***DDHR***)

$$\equiv\ \lambda\eta\,.\,\lambda\mathcal{A}\,.\,x \xleftarrow{\$} \texttt{sample}(q);\ y \xleftarrow{\$} \texttt{sample}(q);\ b \xleftarrow{\$} \texttt{rand};\ \langle m_0, m_1, \mathcal{A}'\rangle \xleftarrow{\$} \mathcal{A}(\eta, \gamma^x);\ v' \xleftarrow{\$} \left(\begin{array}{l} z \xleftarrow{\$} \texttt{sample}(q); \\ \texttt{return}(\gamma^z * m_b) \end{array}\right);\ b' \xleftarrow{\$} \mathcal{A}'(\gamma^y, v');\ \texttt{return}(b \stackrel{?}{=} b')$$

(By the equivalence rules *AX-BIND-3* and *AX-BIND-1* in [36])

$$\approx\ \lambda\eta\,.\,\lambda\mathcal{A}\,.\,x \xleftarrow{\$} \texttt{sample}(q);\ y \xleftarrow{\$} \texttt{sample}(q);\ b \xleftarrow{\$} \texttt{rand};\ \langle m_0, m_1, \mathcal{A}'\rangle \xleftarrow{\$} \mathcal{A}(\eta, \gamma^x);\ v' \xleftarrow{\$} \left(\begin{array}{l} z \xleftarrow{\$} \texttt{sample}(q); \\ \texttt{return}(\gamma^z) \end{array}\right);\ b' \xleftarrow{\$} \mathcal{A}'(\gamma^y, v');\ \texttt{return}(b \stackrel{?}{=} b')$$

(By Lemma 4 as $(_ * m_b)$ is a permutation over the group when m_b is also from the group)

$$\equiv\ \lambda\eta\,.\,\lambda\mathcal{A}\,.\,b \xleftarrow{\$} \texttt{rand};\ x \xleftarrow{\$} \texttt{sample}(q);\ y \xleftarrow{\$} \texttt{sample}(q);\ z \xleftarrow{\$} \texttt{sample}(q);\ \langle m_0, m_1, \mathcal{A}'\rangle \xleftarrow{\$} \mathcal{A}(\eta, \gamma^x);\ b' \xleftarrow{\$} \mathcal{A}'(\gamma^y, \gamma^z);\ \texttt{return}(b \stackrel{?}{=} b')$$

(By the equivalence rules *AX-BIND-3* and *AX-BIND-1* in [36])

$$\equiv\ \lambda\eta\,.\,\lambda\mathcal{A}\,.\,\texttt{rand}$$

(By Lemma 3)

Figure 4: Proof of semantic security of ElGamal

Quadratic Residuosity Assumption. The quadratic residuosity problem is the following: given an odd composite integer n, decide whether or not an $x \in \mathbb{Z}_n^*(+1)$ is a quadratic residue modulo n. The quadratic residuosity assumption (QRA) states that the above problem is intractable when n is the product of two distinct odd primes [26]. We reformulate the assumption in $\mathrm{CSLR}^{\$}_{\pi}$:

$$\lambda\eta\,.\,\lambda\mathcal{A}\,.\,x \xleftarrow{\$} \lceil\mathbb{Z}_n^*(+1)\rceil;\ b \xleftarrow{\$} \mathcal{A}(\eta, n, x);\ \mathtt{return}(b \stackrel{?}{=} \boldsymbol{qr}(x)) \quad \approx \ \lambda\eta\,.\,\lambda\mathcal{A}\,.\,\mathtt{rand}$$

where $\mathcal{A}$ is of type $\square\mathsf{Bits} \to \square\mathsf{Bits} \to \square\mathsf{Bits} \to \mathsf{TBits}$, $\boldsymbol{qr}(x)$ is the quadratic residuosity test of the element x of $\mathbb{Z}_n^*$ in our encoding, and n is a bitstring expression that depends on the security parameter η. $\lceil\mathbb{Z}_n^*(+1)\rceil$ is the $\mathrm{CSLR}^{\$}$ implementation of the uniform distribution $\mathbb{Z}_n^*(+1)$, which can be defined using the uniform sampling primitive `sample`. The function $\boldsymbol{qr}$ is not definable in $\mathrm{CSLR}^{\$}$ and is considered here as a native constant of type[4] $\mathsf{Bits} \multimap \mathsf{Bits}$ in $\mathrm{CSLR}^{\$}_{\pi}$.

Also notice that the integer n, as well as elements in $\mathbb{Z}_n^*$, are polynomially bounded by the security parameter η, so the adversary can safely make recursion over n and elements of $\mathbb{Z}_n^*$ without exceeding the complexity bound. This is often made implicit, sometimes unclear in traditional cryptographic proofs, and can lead to errors in proofs. $\mathrm{CSLR}^{\$}$ makes it explicit by forcing the users to write well-typed terms that can be automatically type-checked.

The proof of left-bit unpredictability of BBS also relies on a set of number-theoretic facts about $\mathbb{Z}_n^*$ and $\mathbb{QR}_n$ (the set of quadratic residues modulo n). We list some of the facts that are necessary for building the proof, and their implications in terms of $\mathrm{CSLR}^{\$}_{\pi}$ programs. These facts (except the first one) assume that n is a Blum integer, which is the product of two distinct primes, both congruent to 3 modulo 4. In this case, each $x \in \mathbb{QR}_n$ has a unique square root in $\mathbb{QR}_n$, which is called the principal square root and denoted by $\sqrt{x}$. Note that in the sequel, we simply write x^2 (omitting the $\bmod n$ part) for the group square operations in $\mathbb{Z}_n^*$.

Fact 1. The function which maps an $x \in \mathbb{Z}_n^*$ to $x^2 \in \mathbb{QR}_n$ is a surjective 4-to-1 function. By Lemma 2, this implies that

$$x \xleftarrow{\$} \lceil\mathbb{Z}_n^*\rceil;\ \mathtt{return}(x^2) \equiv \lceil\mathbb{QR}_n\rceil.$$

Fact 2. The function which maps an $x \in \mathbb{QR}_n$ to $x^2 \in \mathbb{QR}_n$ is a permutation. By Lemma 2, this implies that

$$x \xleftarrow{\$} \lceil\mathbb{QR}_n\rceil;\ \mathtt{return}(x^2) \equiv \lceil\mathbb{QR}_n\rceil.$$

Fact 3. The function which maps an $x \in \mathbb{Z}_n^*(+1)$ to $x^2 \in \mathbb{QR}_n$ is a surjective (2-to-1) function. By Lemma 2, this implies that

$$x \xleftarrow{\$} \lceil\mathbb{Z}_n^*(+1)\rceil;\ \mathtt{return}(x^2) \equiv \lceil\mathbb{QR}_n\rceil.$$

[4]The type of $\boldsymbol{qr}$ is defined as a linear function type so that it can take linear arguments.

Fact 4. For all $x \in \mathbb{QR}_n$, $\sqrt{x^2} = x$.
Fact 5. For all $x \in \mathbb{Z}_n^*(+1)$, $x \in \mathbb{QR}_n \Leftrightarrow \textbf{\textit{parity}}(x) = \textbf{\textit{parity}}(\sqrt{x^2})$.

$\mathrm{CSLR}_\pi^\$$ is expressive enough to encode the proof of [31] that BBS is left-bit unpredictable.

Theorem 2 (Left-bit unpredictability of Blum-Blum-Shub PRG)**.** *For every positive integer l,*

$$\begin{array}{l} \lambda\eta\,.\,\lambda\mathcal{A}\,.\,s \xleftarrow{\$} \lceil \mathbb{Z}_n^* \rceil;\; u \leftarrow \textbf{\textit{BBS}}(\eta, l+1, s); \\ \qquad\qquad b \xleftarrow{\$} \mathcal{A}(\eta, n, \textbf{\textit{tail}}(u));\; \mathtt{return}(b \stackrel{?}{=} \textbf{\textit{head}}(u)) \end{array} \approx \lambda\eta\,.\,\lambda\mathcal{A}\,.\,\mathtt{rand}$$

where n is a Blum integer polynomially bound by the security parameter η and $\mathcal{A}$ is assumed to be a CSLR term of type $\square\mathsf{Bits} \to \square\mathsf{Bits} \to \square\mathsf{Bits} \to \mathsf{TBits}$.

Proof. The proof is done in the proof system of $\mathrm{CSLR}_\pi^\$$ (cf. Figure 5). □

6. Conclusions

We have extended Zhang's CSLR into $\mathrm{CSLR}_\pi^\$$ that provides a uniform framework to define cryptographic constructions, feasible adversaries, security notions, computational assumptions, game transformations, and game-based security proofs. $\mathrm{CSLR}_\pi^\$$ keeps the feature of characterizing PPT adversaries through typing in CSLR but allows users to write security games using a richer language, which is closer to the mathematical language and reduces the programming overhead.

As a future work, it might be interesting to allow arbitrary types in $\mathrm{CSLR}_\pi^\$$ because intermediate games might be easier to write without having to encode everything into bitstrings.

The most immediate direction for future work is to consider more complex examples. We could also consider an implementation of ElGamal that would use BBS as a source for pseudorandom bits. Another possible direction would be to implement $\mathrm{CSLR}_\pi^\$$ (possibly in a proof assistant) and develop a library of reusable security definitions, assumptions and game transformations. This would help dealing with complex examples.

The notion of oracle is frequently used in cryptography and it is sometimes necessary for defining security notions. For instance, with symmetric keys, an encryption oracle allows the adversary to encrypt messages without knowing the key. The higher-order nature of $\mathrm{CSLR}_\pi^\$$ makes it easy to define such oracles and it would be interesting to explore this direction.

Acknowledgements

We would like to thank the reviewers for their comments and suggestions that helped to improve significantly this paper.

$$\lambda\eta\,.\,\lambda\mathcal{A}\,.\,s \stackrel{\$}{\leftarrow} \lceil\mathbb{Z}_n^*\rceil;\ u \leftarrow \boldsymbol{BBS}(\eta, l+1, s);\ b \stackrel{\$}{\leftarrow} \mathcal{A}(\eta, n, \boldsymbol{tail}(u));\ \texttt{return}(b \stackrel{?}{=} \boldsymbol{head}(u))$$

$$\equiv\ \lambda\eta\,.\,\lambda\mathcal{A}\,.\,s \stackrel{\$}{\leftarrow} \lceil\mathbb{Z}_n^*\rceil;\ x \leftarrow s^2;\ u \leftarrow \boldsymbol{parity}(x)\bullet\boldsymbol{bbsrec}(\eta, l, x^2);\ b \stackrel{\$}{\leftarrow} \mathcal{A}(\eta, n, \boldsymbol{tail}(u));\ \texttt{return}(b \stackrel{?}{=} \boldsymbol{head}(u))$$

(Inline of the definition of ***BBS***)

$$\equiv\ \lambda\eta\,.\,\lambda\mathcal{A}\,.\,x \stackrel{\$}{\leftarrow} \left(\begin{array}{l} s \stackrel{\$}{\leftarrow} \lceil\mathbb{Z}_n^*\rceil; \\ \texttt{return}(s^2) \end{array}\right);\ b_0 \leftarrow \boldsymbol{parity}(x);\ v \leftarrow \boldsymbol{bbsrec}(\eta, l, x^2);\ b \stackrel{\$}{\leftarrow} \mathcal{A}(\eta, n, v);\ \texttt{return}(b \stackrel{?}{=} b_0)$$

(By the equivalence rules *AX-BIND-3* and *AX-BIND-1* in [36])

$$\equiv\ \lambda\eta\,.\,\lambda\mathcal{A}\,.\,x \stackrel{\$}{\leftarrow} \lceil\mathbb{QR}\rceil_n;\ b_0 \leftarrow \boldsymbol{parity}(x);\ v \leftarrow \boldsymbol{bbsrec}(\eta, l, x^2);\ b \stackrel{\$}{\leftarrow} \mathcal{A}(\eta, n, v);\ \texttt{return}(b \stackrel{?}{=} b_0)$$

(By Fact 1)

$$\equiv\ \lambda\eta\,.\,\lambda\mathcal{A}\,.\,x \stackrel{\$}{\leftarrow} \lceil\mathbb{QR}\rceil_n;\ b_0 \leftarrow \boldsymbol{parity}(\sqrt{x^2});\ v \leftarrow \boldsymbol{bbsrec}(\eta, l, x^2);\ b \stackrel{\$}{\leftarrow} \mathcal{A}(\eta, n, v);\ \texttt{return}(b \stackrel{?}{=} b_0)$$

(By Fact 4)

$$\equiv\ \lambda\eta\,.\,\lambda\mathcal{A}\,.\,y \stackrel{\$}{\leftarrow} \left(\begin{array}{l} x \stackrel{\$}{\leftarrow} \lceil\mathbb{QR}_n\rceil; \\ \texttt{return}(x^2) \end{array}\right);\ b_0 \leftarrow \boldsymbol{parity}(\sqrt{y});\ v \leftarrow \boldsymbol{bbsrec}(\eta, l, y);\ b \stackrel{\$}{\leftarrow} \mathcal{A}(\eta, n, v);\ \texttt{return}(b \stackrel{?}{=} b_0)$$

(By the equivalence rules *AX-BIND-3* and *AX-BIND-1* in [36])

$$\equiv\ \lambda\eta\,.\,\lambda\mathcal{A}\,.\,y \stackrel{\$}{\leftarrow} \lceil\mathbb{QR}_n\rceil;\ b_0 \leftarrow \boldsymbol{parity}(\sqrt{y});\ v \leftarrow \boldsymbol{bbsrec}(\eta, l, y);\ b \stackrel{\$}{\leftarrow} \mathcal{A}(\eta, n, v);\ \texttt{return}(b \stackrel{?}{=} b_0)$$

(By Fact 2)

$$\equiv\ \lambda\eta\,.\,\lambda\mathcal{A}\,.\,y \stackrel{\$}{\leftarrow} \left(\begin{array}{l} z \stackrel{\$}{\leftarrow} \lceil\mathbb{Z}_n^*(+1)\rceil; \\ \texttt{return}(z^2) \end{array}\right);\ b_0 \leftarrow \boldsymbol{parity}(\sqrt{y});\ v \leftarrow \boldsymbol{bbsrec}(\eta, l, y);\ b \stackrel{\$}{\leftarrow} \mathcal{A}(\eta, n, v);\ \texttt{return}(b \stackrel{?}{=} b_0)$$

(By Fact 3.)

$$\equiv\ \lambda\eta\,.\,\lambda\mathcal{A}\,.\,z \stackrel{\$}{\leftarrow} \lceil\mathbb{Z}_n^*(+1)\rceil;\ b_0 \leftarrow \boldsymbol{parity}(\sqrt{z^2});\ v \leftarrow \boldsymbol{bbsrec}(\eta, l, z^2);\ b \stackrel{\$}{\leftarrow} \mathcal{A}(\eta, n, v);\ \texttt{return}(b \stackrel{?}{=} b_0)$$

(By the equivalence rules *AX-BIND-3* and *AX-BIND-1* in [36].)

$$\equiv\ \lambda\eta\,.\,\lambda\mathcal{A}\,.\,z \stackrel{\$}{\leftarrow} \lceil\mathbb{Z}_n^*(+1)\rceil;\ v \leftarrow \boldsymbol{bbsrec}(\eta, l, z^2);\ b \stackrel{\$}{\leftarrow} \mathcal{A}(\eta, n, v);\ \texttt{return}(b \oplus \boldsymbol{parity}(z) \oplus 1 \stackrel{?}{=} \boldsymbol{parity}(\sqrt{z^2}) \oplus \boldsymbol{parity}(z) \oplus 1)$$

(By the equivalence rule *AX-BIND-1* in [36], and apply function $\lambda x\,.\,x \oplus \boldsymbol{parity}(z) \oplus 1$ to both sides of equality test.)

$$\equiv\ \lambda\eta\,.\,\lambda\mathcal{A}\,.\,z \stackrel{\$}{\leftarrow} \lceil\mathbb{Z}_n^*(+1)\rceil;\ v \leftarrow \boldsymbol{bbsrec}(\eta, l, z^2);\ b \stackrel{\$}{\leftarrow} \mathcal{A}(\eta, n, v);\ \texttt{return}(b \oplus \boldsymbol{parity}(z) \oplus 1 \stackrel{?}{=} \boldsymbol{qr}(z))$$

(By Fact 5, where $\oplus$ is the notation of exclusive-or.)

$$\equiv\ \lambda\eta\,.\,\lambda\mathcal{A}\,.\,z \stackrel{\$}{\leftarrow} \lceil\mathbb{Z}_n^*(+1)\rceil;\ b' \stackrel{\$}{\leftarrow} \left(\begin{array}{l} v \leftarrow \boldsymbol{bbsrec}(\eta, l, z^2); \\ b \stackrel{\$}{\leftarrow} \mathcal{A}(\eta, n, v); \\ \texttt{return}(b \oplus \boldsymbol{parity}(z) \oplus 1) \end{array}\right);\ \texttt{return}(b' \stackrel{?}{=} \boldsymbol{qr}(z))$$

(By the equivalence rules *AX-BIND-3* and *AX-BIND-1* in [36].)

$$\approx\ \lambda\eta\,.\,\lambda\mathcal{A}\,.\,\texttt{rand}$$

(By QRA, considering the adversary $\lambda\eta\,.\,\lambda n\,.\,\lambda x\,.\left(\begin{array}{l} v \leftarrow \boldsymbol{bbsrec}(\eta, l, x^2); \\ b \stackrel{\$}{\leftarrow} \mathcal{A}(\eta, n, v); \\ \texttt{return}(b \oplus \boldsymbol{parity}(x) \oplus 1) \end{array}\right)$, which is PPT by CSLR typing!)

Figure 5: Proof of left-unpredictability of BBS

References

[1] R. Affeldt, D. Nowak, and K. Yamada. Certifying assembly with formal cryptographic proofs: the case of BBS. In *Science of Computer Programming*, 77(10-11):1058–1074, 2012.

[2] M. Backes, M. Berg, and D. Unruh. A formal language for cryptographic pseudocode. In *Proceedings of the 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2008)*, volume 5330 of Lecture Notes in Computer Science, pages 353–376. Springer.

[3] G. Barthe, M. Daubignard, B. Kapron, and Y. Lakhnech. Computational Indistinguishability Logic. In *Proceedings of the 17th ACM conference on Computer and communications security (CCS '10)*, pages 375–386. ACM.

[4] G. Barthe, B. Grégoire, S. Heraud, and S. Zanella Béguelin. Computer-Aided Security Proofs for the Working Cryptographer. In *Proceedings of the 31st Annual International Cryptology Conference (CRYPTO 2011)*, volume 6841 of *Lecture Notes in Computer Science*, pages 71–90. Springer.

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

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

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

[8] B. Blanchet and D. Pointcheval. Automated security proofs with sequences of games. In *Proceedings of the 26th Annual International Cryptology Conference (CRYPTO 2006)*, volume 4117 of *Lecture Notes in Computer Science*, pages 537–554. Springer.

[9] L. Blum, M. Blum, and M. Shub. A simple unpredictable pseudo-random number generator. SIAM Journal on Computing, 15(2):364–383. Society for Industrial and Applied Mathematics, 1986.

[10] D. Boneh. The Decision Diffie-Hellman problem. In *Proceedings of the 3rd International Symposium on Algorithmic Number Theory (ANTS-III)*, volume 1423 of Lecture Notes in Computer Science, pages 48–83. Springer, 1998.

[11] A. Cobham. The intrinsic computational difficulty of functions. In *Proceedings of the 1964 International Congress for Logic, Methodology, and the Philosophy of Science*, pages 24–30. North Holland.

[12] R. Corin and J. den Hartog. A probabilistic Hoare-style logic for game-based cryptographic proofs. In *Proceedings of the 33rd International Colloquium on Automata, Languages and Programming (ICALP 2006)*, volume 4052 of Lecture Notes in Computer Science, pages 252–263. Springer.

[13] J. Courant, M. Daubignard, C. Ene, P. Lafourcade, and Y. Lakhnech. Towards automated proofs for asymmetric encryption schemes in the random oracle model. In *Proceedings of the 15th ACM Conference Computer and Communications Security (CCS 2008)*, pages 371–380. ACM.

[14] U. Dal Lago and P. Parisen Toldin. A Higher Order Characterization of Probabilistic Polynomial Time. In *Draft Proceedings of the 2nd International Workshop on Foundational and Practical Aspects of Resource Analysis (FOPARA 2011)*, Technical Report SIC-08/11, Dept. Computer Systems and Computing, Universidad Complutense de Madrid, pages 1–7.

[15] W. Diffie and M. E. Hellman. New directions in cryptography. In *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[16] T. Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[17] O. Goldreich. *The Foundations of Cryptography: Basic Tools*. Cambridge University Press, 2001.

[18] S. Goldwasser and S. Micali. Probabilistic encryption. In *Journal of Computer and System Sciences (JCSS)*, 28(2):270–299. Academic Press, 1984.

[19] S. Halevi. A plausible approach to computer-aided cryptographic proofs. Cryptology ePrint Archive, Report 2005/181, 2005.

[20] S. Heraud and D. Nowak. A formalization of polytime functions. In *Proceedings of the 2nd International Conference on Interactive Theorem Proving (ITP 2011)*, volume 6898 of Lecture Notes in Computer Science, pages 119–134. Springer.

[21] M. Hofmann. A Mixed Modal/Linear Lambda Calculus with Applications to Bellantoni-Cook Safe Recursion. In *Proceeding of the 11th International Workshop on Computer Science Logic (CSL 1997)*, volume 1414 of Lecture Notes in Computer Science, pages 275–294. Springer.

[22] M. Hofmann. Safe recursion with higher types and BCK-algebra. In *Annals of Pure and Applied Logic*, volume 1414 of 104(1-3):113–166, 2000.

[23] J. Hurd. A formal approach to probabilistic termination. In *Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2002)*, volume 2410 of Lecture Notes in Computer Science, pages 230–245. Springer.

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

[25] J.-Y. Marion. A type system for complexity flow analysis. In *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science (LICS 2011)*, pages 123–132. IEEE Computer Society.

[26] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. Handbook of Applied Cryptography. CRC Press, 1996.

[27] J. C. Mitchell, M. Mitchell, and A. Scedrov. A linguistic characterization of bounded oracle computation and probabilistic polynomial time. In *Proceedings of the 39th Annual Symposium on Foundations of Computer Science (FOCS'98)*, pages 725–733. IEEE Computer Society.

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

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

[30] D. Nowak. A framework for game-based security proofs. In *Proceedings of the 9th International Conference on Information and Communications Security (ICICS 2007)*, volume 4861 of Lecture Notes in Computer Science, pages 319–333. Springer.

[31] D. Nowak. On formal verification of arithmetic-based cryptographic primitives. In *Proceedings of the 11th International Conference on Information Security and Cryptology (ICISC 2008)*, volume 5461 of Lecture Notes in Computer Science, pages 368–382. Springer.

[32] D. Nowak and Y. Zhang. A calculus for game-based security proofs. In *Proceedings of the 4th International Conference on Provable Security (ProvSec 2010)*, volume 6402 of Lecture Notes in Computer Science, pages 35–52. Springer.

[33] N. Ramsey and A. Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *Proceedings of the 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2002)*, pages 154–165.

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

[35] A.C. Yao. Theory and applications of trapdoor functions. In *Proceedings of the IEEE 23rd Annual Symposium on Foundations of Computer Science (FOCS'82)*, pages 80–91. IEEE Computer Society.

[36] Y. Zhang. The Computational SLR: a logic for reasoning about computational indistinguishability. In *Mathematical Structures in Computer Science*, 20(5):951–975, 2010.