A Calculus for Game-based Security Proofs

David Nowak[1] and Yu Zhang[2]

[1] Research Center for Information Security, AIST, Japan
[2] Institute of Software, Chinese Academy of Sciences, China

Abstract. The game-based approach to security proofs in cryptography is a widely-used methodology for writing proofs rigorously. However a unifying language for writing games is still missing. In this paper we show how CSLR, a probabilistic lambda-calculus with a type system that guarantees that computations are probabilistic polynomial time, can be equipped with a notion of game indistinguishability. This allows us to define cryptographic constructions, effective adversaries, security notions, computational assumptions, game transformations, and game-based security proofs in the unified framework provided by CSLR. Our code for cryptographic constructions is close to implementation in the sense that we do not assume primitive uniform distributions but use a realistic algorithm to approximate them. We illustrate our calculus on cryptographic constructions for public-key encryption and pseudorandom bit generation.

Keywords: game-based proofs, implicit complexity, computational indistinguishability

1 Introduction

Cryptographic constructions are fundamental components for information security. A cryptographic construction must come with a security proof. But those proofs can be subtle and tedious, and thus not easy to check. Bellare and Rogaway even claim in [9] that:

> "*Many proofs in cryptography have become essentially unverifiable. Our field may be approaching a crisis of rigor.*"

With Shoup [27], they advocate game-based proofs as a remedy. This is a methodology for writing security proofs that makes them easier to read and check. In this approach, a security property is modeled as a probabilistic program 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 a random player. An adversary is assumed to be efficient i.e., it is modeled as a probabilistic polynomial-time (for short, PPT) function.

However a unifying language for writing games is still missing. In this paper we show how Computational SLR [29] (for short, CSLR), a probabilistic lambda-calculus with a type system that guarantees that computations are probabilistic polynomial time, can be equipped with a notion of game indistinguishability. This allows us to define cryptographic constructions, effective adversaries, security notions, computational assumptions, game transformations, and game-based security proofs in the unified framework provided by CSLR.

Related work. Nowak has given a formal account of the game-based approach, and formalized it in the proof assistant Coq [24, 25]. He follows Shoup by modeling games directly as probability distributions, without going through a programming language. With this approach, he can machine-check game transformations, but not the complexity bound on the adversary. Previously, Corin and den Hartog had proposed a probabilistic Hoare logic in [10] to formalize game-based proofs but they suffer from the same limitation. This issue is addressed in [3] where the authors mention that their

implementation includes tactics that can help establishing that a program is PPT. Their approach is direct in the sense that polynomial-time computation is characterized by explicitly counting the number of computation steps. Backes et al. [2] are also working on a similar approach with the addition of higher-order aimed at reasoning about oracles.

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, by adding a new kind of game transformation (so-called implementation steps), game-based security proofs can be conducted directly on implementations in assembly language. They have applied their approach to the verification of an implementation in assembly language of a pseudorandom bit generator (PRBG). However they do not address the issue of uniform distributions. Indeed, 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 [14]. In the case of another cardinal, the uniform distribution can only either be approximated or rely on code that might not terminate, although it will terminate with a probability equal to 1 [18]. With arbitrary random choices, one can define more distributions than those allowed by the definition of PPT. This raises a fundamental concern that is usually overlooked by cryptographers.

Mitchell et al. have proposed a process calculus with bounded replications and messages to guarantee that those processes are computable in polynomial time [22]. Messages can be terms of OSLR — SLR with a random oracle [21]. 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. Blanchet and Pointcheval have implemented CryptoVerif, a semi-automatic tool for making game-based security proofs, also based on a process calculus. Courant et al. have proposed a specialized Hoare logic for analyzing generic asymmetric encryption schemes in the random oracle model [11]. In our work, we do not want to restrict ourselves to *generic* schemes. Impagliazzo and Kapron have proposed two logics for reasoning about cryptographic constructions [19]. 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. More recently Zhang has developed a logic for computational indistinguishability on top of Hofmann's SLR [29].

Contributions. We propose to use CSLR [29] to conduct game-based security proofs. Because the only basic type in CSLR is the type for bits, our code for cryptographic constructions is closer to implementation than the code in related work: in particular, we address the issue of uniform distributions by using an algorithm that approximates them.

CSLR 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 defined security notions but are not meant to make their way into implementations. We thus extend CSLR into CSLR+ that allows for superpolynomial-time computations and arbitrary uniform choices. However the cryptographic constructions and the adversary will be constrained to be terms of CSLR.

We propose a notion of game indistinguishability. Although, it is not stronger than the notion of computational indistinguishability of [29], 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 [29]. We illustrate the usability of our approach by: proving formally in our proof system for CSLR that an implementation in CSLR of the public-key encryption scheme ElGamal is semantically secure; and by formalizing the pseudorandom bit generator of Blum, Blum and Shub with the related security definition and computational assumption.

Compared with [2] and [3], our approach has the advantage that it can automatically prove (by type inference [16]) that a program is PPT [17].

Outline. We introduce CSLR in Section 2, and use it in Section 3 to define cryptographic constructions. We also discuss in Section 3 the problem of approximating uniform sampling from sets of arbitrary size using just fair coin tosses. In Section 4, we introduce the notion of game indistinguishability that we use to define security notions (using higher-order) and game transformations. We deal with the example of ElGamal in Section 5. In Section 6, we extend CSLR with superpolynomial-time primitives and arbitrary uniform choices to be used in intermediate games, and we illustrate this extension on the pseudorandom bit generator of Blum, Blum and Shub. Finally, we conclude in Section 7.

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 [4]. 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 [16, 17]. 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 [16]. Mitchell et al. have extended SLR by adding a random bit oracle to simulate the oracle tape in probabilistic Turing-machines [21].

More recently, Zhang has introduced CSLR, a non-polymorphic version of SLR extended with probabilistic computations and a primitive notion of bitstrings [29]. His use of monadic types [23], allows for an explicit distinction in CSLR between probabilistic and purely deterministic functions. It was not possible with the extension by Mitchell et al. [21]. We recall below the definition of CSLR and its main property.

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 [17]. $\tau \times \tau'$ 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 3).

SLR also have 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 CSLR, with an additional rule saying that the constructor T preserves sub-typing [29].

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

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

$\mathtt{B}_0$ 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. $\mathtt{case}_\tau$ 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 $f_1(n)$ 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. $\mathtt{rand}$ returns a random bit 0 or 1, each with the probability $\frac{1}{2}$. $\mathtt{return}(e)$ is the trivial (deterministic) computation which returns e with probability 1. $\mathtt{bind}\ x \leftarrow e_1\ \mathtt{in}\ e_2$ 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 [17].

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;
- $x \overset{\$}{\leftarrow} e_1;\ e_2$ represents the probabilistic sequential computation $\mathtt{bind}\ x \leftarrow e_1\ \mathtt{in}\ e_2$;
- $x \leftarrow e_1;\ e_2$ represents the deterministic sequential (call-by-value) computation $(\lambda x.\, e_2) e_1$;
- $\mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2$ represents a simple case distinction $\mathtt{case}(e, \langle e_2, \lambda\_.e_2, \lambda\_.e_1 \rangle)$, 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 $\lambda n.\, \mathtt{rec}_\tau(e_1, e_2, n)$, we often write the definition as:

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

  where $\overset{?}{=}$ 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 [29].

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 [29].

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 [29]. We write $\mathbb{B}$ for the set of bitstrings, with a special element ϵ denoting the empty bitstring. To interpret the probabilistic computations, we adopt the probabilistic monad defined in [26]: if A is set, we write $\mathcal{D}_A : A \to [0, 1]$ for the set of probability mass functions over A. The original monad in [26] is defined using measures instead of mass functions, and is of type $(2^A \to [0, \infty]) \to [0, \infty]$, where 2^A denotes the set of all subsets of A, so that it can also represent computing probabilities over infinite data structure, not just discrete probabilities. But for the sake of simplicity, in this paper as well as in [29] we work on mass functions instead of measures. Note that the monad is not the one defined in [21], 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 [24, 25].

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:

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

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 [29].

The main property of CSLR [21, 29] is:

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

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

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

$$\begin{aligned}\textbf{\textit{rs}} \stackrel{\text{def}}{=} \lambda n\,.\,\texttt{if}\ (n \stackrel{?}{=} \texttt{nil})\ \texttt{then return}(\texttt{nil})\\ \texttt{else}\ b \stackrel{\$}{\leftarrow} \texttt{rand};\ u \stackrel{\$}{\leftarrow} \textbf{\textit{rs}}(\textbf{\textit{tail}}(n));\ \texttt{return}(b \bullet u)\end{aligned}$$

where $\bullet$ denotes the concatenation operation of bitstrings, which can be programmed and typed in CSLR [29]. ***rs*** receives a bitstring and returns a uniformly random bitstring of the same length. It can be checked that $\vdash \textbf{\textit{rs}} : \Box\mathsf{Bits} \to \mathsf{TBits}$.

3 Cryptographic constructions in CSLR

Uniform distributions are ubiquitous in cryptography. However modern computers are based on binary digits, and thus in implementations the cardinal of the support of a uniform distribution has to 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. We implement this pseudo-uniform sampling in CSLR as follows:

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

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 binaries (of the equal length of q) of $\{0, 1, \ldots, \widehat{q} - 1\}$, and $\mathbb{Z}_q^{\$}$ for the truly uniform distribution from $\mathbb{Z}_q$.

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 [13] can be implemented in CSLR by the following programs:

– Key generation:

$$\textbf{\textit{KG}} \stackrel{\text{def}}{=} \lambda\eta\,.\,x \stackrel{\$}{\leftarrow} \textbf{\textit{zrand}}(q, \eta);\ \texttt{return}(\gamma^x, x)$$

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

– Encryption:

$$\textbf{\textit{Enc}} \stackrel{\text{def}}{=} \lambda\eta\,.\,\lambda pk\,.\,\lambda m\,.\,y \stackrel{\$}{\leftarrow} \textbf{\textit{zrand}}(q,\eta);\ \texttt{return}(\gamma^y, pk^y * m)$$

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

– 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 $\square\mathsf{Bits} \to \mathsf{Bits} \to \mathsf{Bits} \to \mathsf{Bits}$, which does not involve monadic type because decryption is deterministic.

Note that when encoding cryptographic constructions in CSLR, we put the security parameter η explicitly as the argument of the programs. However, as we work on bitstrings in CSLR, 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 in $|\eta|$. There are efficient algorithms which computes a suitable DDH group given η, hence can be programmed in CSLR [8].

In the implementation of ***KG*** and ***Enc***, 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., $\textbf{\textit{zrand}}(q, p(\eta))$ where p is a well-typed SLR function of type $\square\mathsf{Bits} \to \mathsf{Bits}$. The choice of p will affect the final distribution of the program and consequently the advantage of adversaries in security games or 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. In this paper, we use the specific timeout for the sake of clarity.

The Blum-Blum-Shub pseudorandom bit generator. The BBS generator defined in [7] is a deterministic function and can be programmed in CSLR as follows:

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

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

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

where n is determined by the security parameter η. ***BBS*** is a well typed SLR-function of type $\square\mathsf{Bits} \to \mathsf{Bits} \to \mathsf{Bits} \to \mathsf{Bits}$, with the second argument being the length of the resulted pseudo-random bitstring and the third argument being the seed.

4 Game indistinguishability

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 CSLR program of type $\square\mathsf{Bits} \to \mathsf{T}\tau$, where the security parameter will bound its running time, and τ is the type of messages returned by the adversary. In CSLR, a *game* is a closed higher-order CSLR function of type $\square\mathsf{Bits} \to (\square\mathsf{Bits} \to \mathsf{T}\tau) \to \mathsf{TBits}$ that returns one bit.

Definition 1 (Game indistinguishability). *Two CSLR games g_1 and g_2 are* game indistinguishable *(written as $g_1 \approx g_2$) if for every term $\mathcal{A}$ such that $\vdash \mathcal{A} : \square\mathsf{Bits} \to \mathsf{T}\tau$, and every positive polynomial P, there exists some $N \in \mathbb{N}$ such that for all bitstring η with $|\eta| \geq N$,*

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

We introduce the notion of game indistinguishability typically for representing game transformations in game-based security proofs. A more general notion of *computational indistinguishability* in cryptography has been defined in the original CSLR system [29].

Definition 2 (Computational indistinguishability [29]). *Two CSLR terms f_1 and f_2, both of type $\square\mathsf{Bits} \to \tau$, are* computationally indistinguishable *(written as $f_1 \simeq f_2$) if for every closed CSLR term $\mathcal{A}$ of type $\square\mathsf{Bits} \to \tau \to \mathsf{TBits}$ and every positive polynomial P, there exists some $N \in \mathbb{N}$ such that for all bitstring η with $|\eta| \geq N$*

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

where ϵ denotes the empty bitstring.

This definition is a reformulation of Definition 3.2.2 of [14] in CSLR. In particular, a CSLR term of type $\square\mathsf{Bits} \to \mathsf{T}\tau$ defines a so-called probabilistic *ensemble*.

Intuitively, the difference between the two notions of indistinguishability is that, computational indistinguishability allows for any arbitrary use of the compared terms by the adversary, while the game indistinguishability provides more control over the adversary as it is usual in game-based security definitions. Hence, game indistinguishability is no stronger than computational indistinguishability as proved in the following proposition. This is why we can sometimes use the CSLR proof system, which is designed for proving computational indistinguishability, for proving game indistinguishability.

Proposition 1. *Computational indistinguishability implies game indistinguishability.*

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

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

Clearly, $\mathbf{Pr}[[\![\mathcal{A}'(\eta, g_i(\eta))]\!] = \texttt{nil}] = \mathbf{Pr}[[\![g_i(\eta, \mathcal{A})]\!] = 1]$, and because g_1 and g_2 are computationally indistinguishable, $\mathbf{Pr}[[\![\mathcal{A}'(\eta, g_1(\eta))]\!] = \texttt{nil}] - \mathbf{Pr}[[\![\mathcal{A}'(\eta, g_2(\eta))]\!] = \texttt{nil}]$ is negligible. □

We will also use the program equivalence defined in [29]. 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.

Our further development in CSLR also relies on the following lemma about ***zrand***:

Lemma 1. *Let q be a CSLR bitstring. The probabilistic ensemble $[\![\lambda\eta\,.\,\textbf{\textit{zrand}}(q, \eta)]\!]$ and the ensemble of truly uniform distributions $\mathbb{Z}_q^{\$}$ are computationally indistinguishable, i.e., for every closed CSLR term $\mathcal{A}$ of type $\square\mathsf{Bits} \to \tau \to \mathsf{TBits}$ and every positive polynomial P, there exists some $N \in \mathbb{N}$ such that for all bitstring η with $|\eta| \geq N$*

$$\left|\mathbf{Pr}[[\![\mathcal{A}(\eta, \textbf{\textit{zrand}}(q, \eta))]\!] \rightsquigarrow \epsilon] - \mathbf{Pr}[[\![\mathcal{A}(\eta)]\!](\mathbb{Z}_q^{\$}) \rightsquigarrow \epsilon]\right| < \frac{1}{P(|\eta|)}.$$

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

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

is negligible with respect to $|\eta|$, where $\varepsilon = \left(\frac{u - \widehat{q}}{u}\right)^{|\eta|}$ and $u = 2^{\lceil \log_2 \widehat{n} \rceil}$. We can then conclude because statistical closeness implies computational indistinguishability (cf. Section 3.2.2 of [14]). □

4.1 Security notions

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. An public-key encryption scheme $(\boldsymbol{KG}, \boldsymbol{Enc}, \boldsymbol{Dec})$ is said to be *semantically secure* [15] if:

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

where $\mathcal{A}$ and $\mathcal{A}'$ are functions of respective types $\square\mathsf{Bits} \to \tau_k \to \mathsf{T}(\tau_m \times \tau_m \times (\tau_e \to \mathsf{TBits}))$ and $\tau_e \to \mathsf{TBits}$. 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 is whether the adversary is right or not.

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

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

where $\mathcal{A}$ is of type $\square\mathsf{Bits} \to \mathsf{Bits} \to \mathsf{Bits}$. Roughly speaking, it means that any adversary $\mathcal{A}$ playing the unpredictability game (left-side game) cannot do significantly better than a random player (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 sequence of bits 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 is whether the adversary is right or not. It was proved by Yao in [28] that left-bit unpredictability is equivalent to passing all polynomial-time statistical tests.

A notion of next-bit unpredictability was defined in [29], 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* if

$$\lambda\eta\,.\,s \xleftarrow{\$} \textbf{\textit{zrand}}(q,\eta);\ \texttt{return}(F(\eta,s)) \quad\simeq\quad \lambda\eta\,.\,s \xleftarrow{\$} \textbf{\textit{zrand}}(q,\eta);\ b \xleftarrow{\$} \texttt{rand};\ \texttt{return}(b \bullet \textbf{\textit{tail}}(F(\eta,s)))$$

Proposition 2. *Strong left-bit unpredictability implies left-bit unpredictability.*

Proof. The proof can be done using the CSLR proof system. See Figure 1 for details. □

$$\lambda\eta\,.\,\lambda\mathcal{A}\,.\ s \xleftarrow{\$} \textbf{\textit{zrand}}(q,\eta);\ u \leftarrow F(\eta,s);\ b \xleftarrow{\$} \mathcal{A}(\eta, \textbf{\textit{tail}}(u));\ \texttt{return}(b \stackrel{?}{=} \textbf{\textit{head}}(u))$$

$$\equiv \lambda\eta\,.\,\lambda\mathcal{A}\,.\ u \xleftarrow{\$} \begin{pmatrix} s \xleftarrow{\$} \textbf{\textit{zrand}}(q,\eta); \\ \texttt{return}(F(\eta,s)) \end{pmatrix};\ b \xleftarrow{\$} \mathcal{A}(\eta, \textbf{\textit{tail}}(u));\ \texttt{return}(b \stackrel{?}{=} \textbf{\textit{head}}(u))$$

(By rules *AX-BIND-3* and *AX-BIND-1* of [29])

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

(By strong left-bit unpredictability)

$$\equiv \lambda\eta\,.\,\lambda\mathcal{A}\,.\ s \xleftarrow{\$} \textbf{\textit{zrand}}(q,\eta);\ b' \xleftarrow{\$} \texttt{rand};\ u \leftarrow b' \bullet \textbf{\textit{tail}}(F(\eta,s));\ b \xleftarrow{\$} \mathcal{A}(\eta, \textbf{\textit{tail}}(u));\ \texttt{return}(b \stackrel{?}{=} \textbf{\textit{head}}(u))$$

(By rules *AX-BIND-3* and *AX-BIND-1* of [29])

$$\equiv \lambda\eta\,.\,\lambda\mathcal{A}\,.\ b' \xleftarrow{\$} \texttt{rand};\ s \xleftarrow{\$} \textbf{\textit{zrand}}(q,\eta);\ u \leftarrow b' \bullet \textbf{\textit{tail}}(F(\eta,s));\ b \xleftarrow{\$} \mathcal{A}(\eta, \textbf{\textit{tail}}(u));\ \texttt{return}(b \stackrel{?}{=} b')$$

(By rules *AX-BIND-3* of [29])

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

(By Lemma 2)

Fig. 1. Proof of Proposition 2

4.2 Game transformations

Game transformation will consist in rewriting modulo the game indistinguishability relation or the computational indistinguishability. In particular, we will reuse as it is the equational proof system of [29] for game transformations.

We will also need some intermediate lemmas. Those lemmas state basic game transformations used in almost all 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 2. *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 $x, b \notin \mathrm{dom}(\Gamma)$.

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.

Lemma 3. *Let f, f' be two closed CSLR terms of type $\Box\mathsf{Bits} \to \mathsf{Bits}$ 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*

$$\lambda\eta\,.\,x \xleftarrow{\$} \textbf{\textit{zrand}}(q,\eta);\ \mathtt{return}(fx) \simeq \lambda\eta\,.\,x \xleftarrow{\$} \textbf{\textit{zrand}}(q,\eta);\ \mathtt{return}(f'(fx))$$

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 $\mathrm{dom}([\![e_1]\!](\eta))$. For every CSLR adversary $\mathcal{A}$ of type $\Box\mathsf{Bits} \to \mathsf{TBits} \to \mathsf{TBits}$, define two new adversaries

$$\mathcal{A}_1 \stackrel{\mathrm{def}}{=} \lambda\eta\,.\,\lambda w\,.\,\mathcal{A}(\eta, x \xleftarrow{\$} w;\ \mathtt{return}(fx))$$
$$\mathcal{A}_2 \stackrel{\mathrm{def}}{=} \lambda\eta\,.\,\lambda w\,.\,\mathcal{A}(\eta, x \xleftarrow{\$} w;\ \mathtt{return}(f'(fx))).$$

Clearly, both $\mathcal{A}_1$ and $\mathcal{A}_2$ are well-typed CSLR adversaries, and $[\![\mathcal{A}(\eta, e_i(\eta))]\!] = [\![\mathcal{A}_i(\eta, \textbf{\textit{zrand}}(q,\eta))]\!]$ $(i = 1, 2)$. According to Lemma 1,

$$\varepsilon_i = |\mathbf{Pr}[[\![\mathcal{A}_i(\eta, \textbf{\textit{zrand}}(q,\eta))]\!] \rightsquigarrow \epsilon] - \mathbf{Pr}[[\![\mathcal{A}_i(\eta)]\!](\mathbb{Z}_q^\$) \rightsquigarrow \epsilon]|$$

$(i = 1, 2)$ are negligible. Also, by Lemma 3.1 of [25], $[\![\mathcal{A}_1(\eta)]\!](\mathbb{Z}_q^\$) = [\![\mathcal{A}_2(\eta)]\!](\mathbb{Z}_q^\$)$ as $[\![f']\!]$ is a permutation. Hence,

$$\begin{aligned}
&|\mathbf{Pr}[[\![\mathcal{A}(\eta, e_1\eta)]\!] \rightsquigarrow \epsilon] - \mathbf{Pr}[[\![\mathcal{A}(\eta, e_2\eta)]\!] \rightsquigarrow \epsilon]| \\
=&|\mathbf{Pr}[[\![\mathcal{A}_1(\eta, \textbf{\textit{zrand}}(q,\eta))]\!] \rightsquigarrow \epsilon] - \mathbf{Pr}[[\![\mathcal{A}_1(\eta)]\!](\mathbb{Z}_q^\$) \rightsquigarrow \epsilon] \\
&-(\mathbf{Pr}[[\![\mathcal{A}_2(\eta, \textbf{\textit{zrand}}(q,\eta))]\!] \rightsquigarrow \epsilon] - \mathbf{Pr}[[\![\mathcal{A}_2(\eta)]\!](\mathbb{Z}_q^\$) \rightsquigarrow \epsilon])| \\
\leq&\varepsilon_1 + \varepsilon_2
\end{aligned}$$

is still negligible. □

5 Applications

5.1 Computational assumptions

Computational assumptions can be defined in CSLR too. As in the case of defining El-Gamal encryption scheme in CSLR, we have to replace all occurrences of uniform distributions by calls to the function ***zrand***.

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 [12] 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 number such that $0 \leq x, y, z < \widehat{q}$[3]. DDH cannot be written directly in CSLR because it involves arbitrary uniform distributions. Instead we write the following assumption that we call *DDH-Bits*:

$$\textbf{\textit{DDHBL}} \simeq \textbf{\textit{DDHBR}}$$

where

$$\textbf{\textit{DDHBL}} \stackrel{\text{def}}{=} \lambda\eta\,.\, x \stackrel{\$}{\leftarrow} \textbf{\textit{zrand}}(q,\eta);\ y \stackrel{\$}{\leftarrow} \textbf{\textit{zrand}}(q,\eta);\ \texttt{return}(\gamma^x, \gamma^y, \gamma^{xy})$$
$$\textbf{\textit{DDHBR}} \stackrel{\text{def}}{=} \lambda\eta\,.\, x \stackrel{\$}{\leftarrow} \textbf{\textit{zrand}}(q,\eta);\ y \stackrel{\$}{\leftarrow} \textbf{\textit{zrand}}(q,\eta);\ z \stackrel{\$}{\leftarrow} \textbf{\textit{zrand}}(q,\eta);\ \texttt{return}(\gamma^x, \gamma^y, \gamma^z)$$

Proposition 3. *DDH-bits holds when the DDH assumption holds.*

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 $\mathrm{dom}([\![e_1]\!](\eta))$. For every CSLR adversary $\mathcal{A}$ of type $\Box\mathsf{Bits} \to \mathsf{TBits} \to \mathsf{TBits}$, define two new adversaries

$$\mathcal{A}_1 \stackrel{\text{def}}{=} \lambda\eta\,.\,\lambda w\,.\,\mathcal{A}(\eta, x \stackrel{\$}{\leftarrow} w;\ \texttt{return}(fx))$$
$$\mathcal{A}_2 \stackrel{\text{def}}{=} \lambda\eta\,.\,\lambda w\,.\,\mathcal{A}(\eta, x \stackrel{\$}{\leftarrow} w;\ \texttt{return}(f'(fx))).$$

Clearly, both $\mathcal{A}_1$ and $\mathcal{A}_2$ are well-typed CSLR adversaries, and $[\![\mathcal{A}(\eta, e_i(\eta))]\!] = [\![\mathcal{A}_i(\eta, \textbf{\textit{zrand}}(q,\eta))]\!]$ $(i = 1, 2)$. According to Lemma 1,

$$\varepsilon_i = |\mathbf{Pr}[[\![\mathcal{A}_i(\eta, \textbf{\textit{zrand}}(q,\eta))]\!] \rightsquigarrow \epsilon] - \mathbf{Pr}[[\![\mathcal{A}_i(\eta)]\!](\mathbb{Z}_q^{\$}) \rightsquigarrow \epsilon]|$$

$(i = 1, 2)$ are negligible. Also, by Lemma 3.1 of [25], $[\![\mathcal{A}_1(\eta)]\!](\mathbb{Z}_q^{\$}) = [\![\mathcal{A}_2(\eta)]\!](\mathbb{Z}_q^{\$})$ as $[\![f']\!]$ is a permutation. Hence,

$$\begin{aligned}
&|\mathbf{Pr}[[\![\mathcal{A}(\eta, e_1\eta)]\!] \rightsquigarrow \epsilon] - \mathbf{Pr}[[\![\mathcal{A}(\eta, e_2\eta)]\!] \rightsquigarrow \epsilon]| \\
=&|\mathbf{Pr}[[\![\mathcal{A}_1(\eta, \textbf{\textit{zrand}}(q,\eta))]\!] \rightsquigarrow \epsilon] - \mathbf{Pr}[[\![\mathcal{A}_1(\eta)]\!](\mathbb{Z}_q^{\$}) \rightsquigarrow \epsilon] \\
&-(\mathbf{Pr}[[\![\mathcal{A}_2(\eta, \textbf{\textit{zrand}}(q,\eta))]\!] \rightsquigarrow \epsilon] - \mathbf{Pr}[[\![\mathcal{A}_2(\eta)]\!](\mathbb{Z}_q^{\$}) \rightsquigarrow \epsilon])| \\
\leq&\varepsilon_1 + \varepsilon_2
\end{aligned}$$

is still negligible. □

[3] We do not assume that $\widehat{q}$ is prime. However most groups in which DDH is believed to be true have prime order [8].

5.2 Semantic security of El-Gamal encryption scheme

In this section, we illustrate our proof system by proving the semantic security of El-Gamal encryption scheme in Fig. 2. The proof follows the same structure as the one in [24], but here the type system of CSLR guarantees that the adversary is probabilistic polynomial-time. This was not dealt with in [24]. Moreover here all transformations are purely syntactic (thus allowing the immediate prospect of being implemented in a tool), while in [24] they were done at the semantics level.

Note that by using Lemma 3, 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 El-Gamal 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 CSLR proof system. Another possibility would be to use a richer type system to reject adversaries returning junk.

6 Extending CSLR

The discussion in the previous sections was limited to the setting of CSLR with bitstrings. In particular, it does not allow superpolynomial-time computations nor arbitrary uniform sampling. Although these restrictions make 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.

In this section, we extend CSLR into CSLR+ so that we can manipulate games with superpolynomial-time computations and arbitrary uniform choices.

6.1 CSLR+

CSLR+ extends CSLR with a uniform sampling primitive `sample` of type $\mathsf{Bits} \multimap \mathsf{TBits}$ and constants for primitive (and possibly superpolynomial-time) computations. `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. For instance, the distribution produced by `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, but we shall omit the implementation details and write $x \xleftarrow{\$} A$; for assigning to x a uniformly sampled value from set A.

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. In fact, in CSLR+ we do not care any more about the complexity class that can be characterized using the type system[4] — CSLR+ is the language for describing games, not adversaries.

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

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

$\equiv \lambda\eta\,.\,\lambda\mathcal{A}\,.\,\langle pk, sk\rangle \xleftarrow{\$} \begin{pmatrix} x \xleftarrow{\$} \textbf{\textit{zrand}}(q, \eta); \\ \mathtt{return}(\gamma^x, x) \end{pmatrix};\ \langle m_0, m_1, \mathcal{A}'\rangle \xleftarrow{\$} \mathcal{A}(\eta, pk);$
$b \xleftarrow{\$} \mathtt{rand};\ c \xleftarrow{\$} \begin{pmatrix} y \xleftarrow{\$} \textbf{\textit{zrand}}(q, \eta); \\ \mathtt{return}(\gamma^y, pk^y * m_b) \end{pmatrix};\ b' \xleftarrow{\$} \mathcal{A}'(c);$
$\mathtt{return}(b \stackrel{?}{=} b')$
(Inline of definition of ***KG*** and ***Enc***)

$\equiv \lambda\eta\,.\,\lambda\mathcal{A}\,.\,x \xleftarrow{\$} \textbf{\textit{zrand}}(q, \eta);\ y \xleftarrow{\$} \textbf{\textit{zrand}}(q, \eta);\ b \xleftarrow{\$} \mathtt{rand};$
$\langle m_0, m_1, \mathcal{A}'\rangle \xleftarrow{\$} \mathcal{A}(\eta, \gamma^x);\ b' \xleftarrow{\$} \mathcal{A}'(\gamma^y, (\gamma^x)^y * m_b);$
$\mathtt{return}(b \stackrel{?}{=} b')$
(By the equivalence rules *AX-BIND-3* and *AX-BIND-1* in [29])

$\equiv \lambda\eta\,.\,\lambda\mathcal{A}\,.\,v \xleftarrow{\$} \textbf{\textit{DDHBL}}(\eta);\ b \xleftarrow{\$} \mathtt{rand};\ \langle m_0, m_1, \mathcal{A}'\rangle \xleftarrow{\$} \mathcal{A}(\eta, \mathtt{proj}_1(v));$
$b' \xleftarrow{\$} \mathcal{A}'(\mathtt{proj}_2(v), \mathtt{proj}_3(v) * m_b);$
$\mathtt{return}(b \stackrel{?}{=} b')$
(Inline of ***DDHBL***)

$\approx \lambda\eta\,.\,\lambda\mathcal{A}\,.\,v \xleftarrow{\$} \textbf{\textit{DDHBR}}(\eta);\ b \xleftarrow{\$} \mathtt{rand};\ \langle m_0, m_1, \mathcal{A}'\rangle \xleftarrow{\$} \mathcal{A}(\eta, \mathtt{proj}_1(v));$
$b' \xleftarrow{\$} \mathcal{A}'(\mathtt{proj}_2(v), \mathtt{proj}_3(v) * m_b);$
$\mathtt{return}(b \stackrel{?}{=} b')$
(By DDH-Bits assumption and *SUB*)

$\equiv \lambda\eta\,.\,\lambda\mathcal{A}\,.\,x \xleftarrow{\$} \textbf{\textit{zrand}}(q, \eta);\ y \xleftarrow{\$} \textbf{\textit{zrand}}(q, \eta);\ z \xleftarrow{\$} \textbf{\textit{zrand}}(q, \eta);\ b \xleftarrow{\$} \mathtt{rand};$
$\langle m_0, m_1, \mathcal{A}'\rangle \xleftarrow{\$} \mathcal{A}(\eta, \gamma^x);\ b' \xleftarrow{\$} \mathcal{A}'(\gamma^y, \gamma^z * m_b);$
$\mathtt{return}(b \stackrel{?}{=} b')$
(Inline of ***DDHBR***)

$\equiv \lambda\eta\,.\,\lambda\mathcal{A}\,.\,x \xleftarrow{\$} \textbf{\textit{zrand}}(q, \eta);\ y \xleftarrow{\$} \textbf{\textit{zrand}}(q, \eta);\ b \xleftarrow{\$} \mathtt{rand};\ \langle m_0, m_1, \mathcal{A}'\rangle \xleftarrow{\$} \mathcal{A}(\eta, \gamma^x);$
$v' \xleftarrow{\$} \begin{pmatrix} z \xleftarrow{\$} \textbf{\textit{zrand}}(q, \eta); \\ \mathtt{return}(\gamma^z * m_b) \end{pmatrix};\ b' \xleftarrow{\$} \mathcal{A}'(\gamma^y, v');$
$\mathtt{return}(b \stackrel{?}{=} b')$
(By the equivalence rules *AX-BIND-3* and *AX-BIND-1* in [29])

$\approx \lambda\eta\,.\,\lambda\mathcal{A}\,.\,x \xleftarrow{\$} \textbf{\textit{zrand}}(q, \eta);\ y \xleftarrow{\$} \textbf{\textit{zrand}}(q, \eta);\ b \xleftarrow{\$} \mathtt{rand};\ \langle m_0, m_1, \mathcal{A}'\rangle \xleftarrow{\$} \mathcal{A}(\eta, \gamma^x);$
$v' \xleftarrow{\$} \begin{pmatrix} z \xleftarrow{\$} \textbf{\textit{zrand}}(q, \eta); \\ \mathtt{return}(\gamma^z) \end{pmatrix};\ b' \xleftarrow{\$} \mathcal{A}'(\gamma^y, v');$
$\mathtt{return}(b \stackrel{?}{=} b')$
(By Lemma 3 as $(\_ * m_b)$ is a permutation over the group when m_b is also from the group)

$\equiv \lambda\eta\,.\,\lambda\mathcal{A}\,.\,b \xleftarrow{\$} \mathtt{rand};\ x \xleftarrow{\$} \textbf{\textit{zrand}}(q, \eta);\ y \xleftarrow{\$} \textbf{\textit{zrand}}(q, \eta);\ z \xleftarrow{\$} \textbf{\textit{zrand}}(q, \eta);$
$\langle m_0, m_1, \mathcal{A}'\rangle \xleftarrow{\$} \mathcal{A}(\eta, \gamma^x);\ b' \xleftarrow{\$} \mathcal{A}'(\gamma^y, \gamma^z);$
$\mathtt{return}(b \stackrel{?}{=} b')$
(By the equivalence rules *AX-BIND-3* and *AX-BIND-1* in [29])

$\equiv \lambda\eta\,.\,\lambda\mathcal{A}\,.\,\mathtt{rand}$
(By Lemma 2)

Fig. 2. Proof of semantic security of ElGamal

The definitions of *computational indistinguishability* and *game indistinguishability* are almost the same as before, except that we are now considering distributions that are produced by CSLR+ programs:

Definition 3 (Game indistinguishability in CSLR+). *Two closed CSLR+ programs g_1 and g_2, both of type $\Box\mathsf{Bits} \to (\Box\mathsf{Bits} \to \mathsf{T}\tau) \to \mathsf{TBits}$, are* game indistinguishable *(written as $g_1 \approx_+ g_2$) if for every closed CSLR term $\mathcal{A}$ 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 bitstring η with $|\eta| \geq N$,*

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

Definition 4 (Comput. indistinguishability in CSLR+). *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 $\Box\mathsf{Bits} \to \tau \to \mathsf{TBits}$ and every positive polynomial P, there exists some $N \in \mathbb{N}$ such that for all bitstring η with $|\eta| \geq N$*

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

where ϵ denotes the empty bitstring.

CSLR+ inherits most of the equational proof system of CSLR. All the rules for program equivalence in CSLR can be used directly in CSLR+. No extra rules are needed for the primitive `sample`, but we can add rules for constants if necessary. The four rules for proving computational indistinguishability remain the same as in CSLR (Figure 3) except that in the rule *SUB*, a new premise enforces that the substitution context (the term e) must be definable in CSLR, i.e., a program that does not contain `sample` or any CSLR+ constant. The soundness of the system still holds and the proof just goes as for CSLR [29]. In particular, the proof for the rule *SUB* contains a construction of a new adversary with the context, which remains a CSLR term (i.e., a PPT adversary) thanks to the new premise enforcing that the context must be definable in CSLR.

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

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

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

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

Fig. 3. Rules for computational indistinguishability in CSLR+

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 explanation and examples in [29].

6.2 Applications

This extension of CSLR+ allows us to express directly DDH in the formalism and thus does not require to go through the non-standard computational assumption introduced in Section 5. We can reproduce almost as such the proof of semantic security for ElGamal given in [24]. The difference is that now we can check automatically that the adversary built in the proof is PPT, and all transformations are purely syntactic.

We can also reproduce the proof of unpredictability for the pseudorandom bit generator of Blum, Blum and Shub given in [25]. The proof requires a test for quadratic residuosity which is a superpolynomial-time computation — it can be introduced into CSLR+ as a constant. Moreover this proof is based on the Quadratic Residuosity Assumption that uses arbitrary uniform choices.

Quadratic Residuosity Assumption. 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. 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 [20]. We reformulate the assumption in CSLR+:

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

where $\mathcal{A}$ must be definable in CSLR of type $\Box\mathsf{Bits} \to \mathsf{Bits} \to \mathsf{Bits} \to \mathsf{TBits}$, $\boldsymbol{qr}(x)$ is the quadratic residuosity test of the element x of $\mathbb{Z}_n^*$ in our encoding, and n is an expression that depends on the security parameter η.

Blum-Blum-Shub. CSLR+ is expressive enough to encode the proof of [25] that BBS is left-bit unpredictable: for every positive integer l,

$$\begin{array}{l}\lambda\eta\,.\,\lambda\mathcal{A}\,.\,s \xleftarrow{\$} \mathbb{Z}_q^*;\ u \leftarrow \boldsymbol{BBS}(\eta, l+1, s);\\ \qquad\qquad b \xleftarrow{\$} \mathcal{A}(\eta, q, \boldsymbol{tail}(u));\ \mathtt{return}(b \stackrel{?}{=} \boldsymbol{head}(u))\end{array} \quad \approx_+ \quad \lambda\eta\,.\,\lambda\mathcal{A}\,.\,\mathtt{rand}$$

where $\mathcal{A}$ is must be definable in CSLR of type $\Box\mathsf{Bits} \to \mathsf{Bits} \to \mathsf{Bits} \to \mathsf{TBits}$.

7 Conclusions

We have shown how Zhang's CSLR can be equipped with a notion of game indistinguishability. The system allows us to define cryptographic constructions, effective adversaries, security notions, computational assumptions, game transformations, and game-based security proofs in the unified framework provided by CSLR. We have illustrated our calculus by formalizing the proof of semantic security for a binary implementation of the public-key encryption scheme ElGamal.

CSLR pushes users to write binary encodings of cryptographic constructions, which are close to their computer implementations, but the programming overhead is probably heavy for people who want to check their cryptographic proofs in the mathematical setting only. Also, the lack of superpolynomial-time computation power limits the application of CSLR in cryptography. We have thus introduced CSLR+ — an extension of CSLR with arbitrary uniform sampling and superpolynomial-time constants, and formalized the pseudorandom bit generator of Blum, Blum

and Shub with the related security definition and computational assumption. CSLR+ 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 CSLR+ 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 El-Gamal that would use BBS as a source for pseudorandom bits. Another possible direction would be to implement CSLR and CSLR+ (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 attacker to encrypt messages without knowing the key. The higher-order nature of CSLR makes it easy to define such oracles. As an example, we can define the notion of *IND-CPA (indistinguishability under chosen plain-text attack)* using encryption oracles: an encryption scheme $(\boldsymbol{KG}, \boldsymbol{Enc}, \boldsymbol{Dec})$ is said to be IND-CPA secure if

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

This is a CSLR reformulation of the definition from [5] (adapted for asymmetric encryption): the game first generates a pair of public and secret keys and a challenge bit b, then sets up a left-right encryption oracle which, upon receiving a pair of messages, will encrypt one of them (according to the challenge bit) using the public key and return the encrypted cipher-text; the public key is passed to the adversary, who is allowed to query the encryption oracle; in CSLR the oracle can be encoded as a function and passed to the adversary as an argument, just as the public key; the adversary then outputs its guess on the challenge b.

The exact relation between these different definitions of security notions remains to be clarified. It would be interesting to investigate how much CSLR can help dealing with oracles.

References

1. R. Affeldt, D. Nowak, and K. Yamada. Certifying assembly with formal cryptographic proofs: the case of BBS. In Proceedings of the 9th International Workshop on Automated Verification of Critical Systems (AVoCS 2009). To appear.
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, 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 Press.
4. S. Bellantoni and S. A. Cook. A new recursion-theoretic characterization of the polytime functions. In *Computational Complexity*, 2:97–110, 1992.
5. M. Bellare and C. Namprempre. Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm. *Journal of Cryptology*, 21:469–491, 2008.

6. 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.
7. Blum, L., Blum, M., Shub, M.: A simple unpredictable pseudo random number generator. SIAM Journal on Computing, 15(2):364–383. Society for Industrial and Applied Mathematics, 1986.
8. 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.
9. M. Bellare and P. Rogaway. Code-based game-playing proofs and the security of triple encryption. Cryptology ePrint Archive, Report 2004/331, 2004.
10. 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.
11. 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 Press.
12. W. Diffie and M. E. Hellman. New directions in cryptography. In *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.
13. 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.
14. O. Goldreich. *The Foundations of Cryptography: Basic Tools*. Cambridge University Press, 2001.
15. S. Goldwasser and S. Micali. Probabilistic encryption. In *Journal of Computer and System Sciences (JCSS)*, 28(2):270–29 9. Academic Press, 1984. An earlier version appeared in *proceedings of STOC'82*.
16. Martin 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)*, volune 1414 of Lecture Notes in Computer Science, pages 275–294. Springer.
17. 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.
18. 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.
19. R. Impagliazzo and B. M. Kapron. Logics for reasoning about cryptographic constructions. *Journal of Computer and System Sciences*, 72(2):286–320, 2006.
20. A. J. Menezes, P. C. van Oorschot, S. A. Vanstone. Handbook of Applied Cryptography. CRC Press, 1996.
21. 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.
22. 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.
23. E. Moggi. Notions of computation and monads. In *Information and Computation*, 93(1):55–92, 1991.
24. 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.
25. 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.
26. 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.
27. V. Shoup. Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332, 2004.
28. 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, 1982.
29. Y. Zhang. The Computational SLR: A Logic for Reasoning about Computational Indistinguishability. In *Proceedings of the 9th International Conference on Typed Lambda Calculi and Applications (TLCA 2009)*, volume 5608 of Lecture Notes in Computer Science, pages 401–415. Springer.