Formal Proof of Provable Security by Game-playing in a Proof Assistant*

Reynald Affeldt[1], Miki Tanaka[2], and Nicolas Marti[3]

[1] Research Center for Information Security,
National Institute of Advanced Industrial Science and Technology
[2] Information Security Research Center,
National Institute of Information and Communications Technology
[3] Department of Computer Science, University of Tokyo

Abstract. Game-playing is an approach to write security proofs that are easy to verify. In this approach, security definitions and intractable problems are written as programs called games and reductionist security proofs are sequences of game transformations. This bias towards programming languages suggests the implementation of a tool based on compiler techniques (syntactic program transformations) to build security proofs, but it also raises the question of the soundness of such a tool. In this paper, we advocate the formalization of game-playing in a proof assistant as a tool to build security proofs. In a proof assistant, starting from just the formal definition of a probabilistic programming language, all the properties required in game-based security proofs can be proved internally as lemmas whose soundness is ensured by proof theory. Concretely, we show how to formalize the game-playing framework of Bellare and Rogaway in the Coq proof assistant, how to prove formally reusable lemmas such as the fundamental lemma of game-playing, and how to use them to formally prove the PRP/PRF Switching Lemma.

1 Introduction

Game-playing is an approach to write security proofs that are easy to verify. In this approach, security definitions and intractable problems are written as programs called games and reductionist security proofs are sequences of game transformations [6–8].

The bias of game-playing towards programming languages suggests the implementation of a tool based on compiler techniques to build security proofs [9], but it also raises the question of the soundness of such a tool. To make our point clearer, let us consider CryptoVerif [13], a pioneer implementation of game-playing that has been applied to several standard cryptographic schemes taken from the literature [4, 5]. To perform game transformations, CryptoVerif implements techniques of compiler optimization (constant propagation, dead-code

* To appear in the proceedings of the 1st International Conference on Provable Security (Provsec) 2007, Wollongong, NSW, Australia, 1–2 November, 2007.

elimination, etc.). The latter program transformations sometimes rely on high-level program equivalences that are only proved on paper and introduced in CryptoVerif as axioms (see Appendix B of [13]). This can be seen as an important limitation of CryptoVerif because it endangers its soundness.

In this paper, we advocate the formalization of game-playing in a proof assistant as a tool to build security proofs. In a proof assistant, starting from just the formal definition of a probabilistic programming language, all the properties required in game-based security proofs can be proved internally as lemmas whose soundness is ensured by proof theory: no game transformation needs to be proved out of the box. Concretely, we show how to formalize the game-playing framework of Bellare and Rogaway [7] in the Coq proof assistant [1], how to prove formally reusable lemmas such as the fundamental lemma of game-playing, and how to use these lemmas to formally prove the PRP/PRF Switching Lemma. To our knowledge, this is the first formalization of game-playing with a random oracle and a working fundamental lemma used in a complete use-case.

About the Coq Proof Assistant The Coq proof assistant [1] is an implementation of proof theory developed at INRIA in France since 1984. It provides a higher-order logic (i.e., even predicates can be quantified) to state mathematical properties and a functional programming language to build proofs. This setting stems from the Curry-Howard isomorphism [2], through which logical formulas are considered as types of functional programs that are themselves considered as proofs. This makes up for a very small and well-understood proof-checking mechanism that justifies the reliability of proof assistants. Proof assistants are now reasonably mature tools and, in particular, the Coq proof assistant recently made it possible for several important achievements such as the formalization of the four color theorem or the certification of a C compiler.

Notations in this Paper All the definitions and lemmas in this paper are written in the Coq syntax. This syntax uses only ASCII characters; the mathematical notations are just to ease reading (for example, we write $\forall$ instead of the Coq `forall` construct, $\wedge$ instead of `/\`, etc.). We display Coq code as it appears in our formalization. To improve understanding, we sometimes put comments (between `(*` and `*)`) or hide non-relevant parts (using "..."). In our experience, using the Coq syntax in this way is the best way to present a formalization because it avoids ambiguities while being accessible to readers with little familiarity with formal methods or functional programming languages. There are some Coq-specific constructs, but we introduce them gently in the first sections. We concentrate on the main points of the formalization (basic definitions and statements of lemmas) and do not enter the details of formal proofs; for technical inquiries, the complete Coq development is available online [16].

The rest of this paper is organized as follows. In Sect. 2, we explain how we formalize the notions of distribution and probability in Coq. In Sect. 3, we explain how we formalize random oracles and a probabilistic programming language to write games. In Sect. 4, we formalize a version of the fundamental lemma of game-playing, the most important tool for game-playing. In Sect. 5, we apply our formalization of the game-playing framework to the proof of the

PRP/PRF Switching Lemma. We review related work in Sect. 6 and conclude in Sect. 7.

2 Formalization of Probabilities

In this section, we explain how we formalize the notion of distribution of states and the notion of probability. We consider a probability space made of *deterministic states* and we call *probabilistic state* a distribution of deterministic states. As far as the formalization of distributions is concerned, we do not commit ourselves to any particular kind of states (this makes our formalization more reusable). In type parlance, we just assume that all deterministic states belong to some type `A` and pursue formalization using this type. In Coq, types themselves have types, and we declare the type `A` to belong to the predefined type `Set` of data structures. This is achieved by the declaration `Variable A : Set.`

In later sections, we instantiate the type `A` with a concrete notion of deterministic state. For example, let us assume that deterministic states are *stores*, i.e., sets of pairs of variables and values. We can use the standard Coq natural numbers (type `nat` of type `Set`) to formalize variables and values, and Coq lists (type `list` of type `Set`) to formalize stores. Variables are defined to be naturals using the definition `Definition var := nat.` Stores are defined to be lists of pairs of variables and naturals by `Definition store := list (var * nat).` Since lists belong to `Set`, so do stores, and therefore we can substitute `A` for `store` to obtain a formalization of distributions of stores.

2.1 Formalization of Distributions

A distribution of deterministic states is a map from deterministic states to real numbers, such that the real numbers associated with a given deterministic state represent the weight of this state in the map. Given the Coq reals (type `R`) and our type `A` of deterministic states, we define distributions to be lists of appropriate type: `Definition distrib := list (R * A).`

There is little point in having distributions with deterministic states associated with negative or null reals. The Coq way to enforce this is to introduce a logical predicate to sort out devious distributions. Like there is the predefined type `Set` for data structures, there is the predefined type `Prop` for logical predicates. Logical predicates in Coq are just definitions with type `Prop`. A logical predicate that holds only for distributions with strictly positive reals (hereafter, *coefficients*) has to go recursively through the underlying list to check the sign of coefficients. Such recursive definitions are introduced by the keyword `Fixpoint`:

```
(* A distribution d has positive coefficients... *)
Fixpoint coeff_pos (d : distrib) : Prop :=
  match d with
    (* if its head and its tail have positive coefficients... *)
    | (p, _) :: tl => 0 < p ∧ coeff_pos tl
    (* or if it is empty.*)
    | nil => True
  end.
```

In our formalization, we do not insist on having the sum of coefficients of distributions equal to 1, as it is customary with probabilities. We made this choice for convenience because of if-then-else constructs in the language of games. Even if we insisted on normalizing probabilities, as soon as the control-flow enters a branch, the distribution is partitioned, and the sum of coefficients in each branch cannot be guaranteed to be equal to the sum of coefficients before the branch (this observation is made in [12]). We therefore express probabilities with respect to the sum of coefficients of the very first distribution. Sums of coefficients are computed with the following recursive function:

```
Fixpoint sum (d : distrib) : R :=
  match d with (p, _) :: tl => p + sum tl | nil => 0 end.
```

2.2 Probability of Events

We identify events with boolean functions over deterministic states, that is to say `Definition event := A → bool.` The probability that an event holds in a distribution is equal to the sum of the coefficients associated with the deterministic states in which this event holds. To sort out relevant deterministic states, we use a function `filter` that selects only those states such that the event `e` holds (`++` is the notation for the Coq function that appends lists):

```
Fixpoint filter (e : event) (d : distrib) : distrib :=
  match d with
    | (p, a) :: tl => (if e a then (p, a) :: nil else nil) ++ filter e tl
    | nil => nil
  end.
```

The probability `Pr` that an event `e` holds in a distribution `d` immediately follows from the definition of `sum` and `filter`:

```
Definition Pr (e : event) (d : distrib) : R := sum (filter e d).
```

Equipped with above definitions of distributions, events and probabilities, we can define concrete events (using Coq standard boolean functions `orb`, `andb`, `negb`, etc.) and prove formally well-known facts in probability theory:

```
(* standard definitions *)
Definition union (e1 e2 : event) (a : A) : event := orb (e1 a) (e2 a).
  (* Notation: _ ∪ _ *)
Definition inter (e1 e2 : event) (a : A) : event := andb (e1 a) (e2 a).
  (* Notation: _ ∩ _ *)
Definition cplt (e : event) (a : A) : event := negb (e a).
  (* Notation: ¬ *)
...
(* well-known facts *)
Lemma Pr_union_inter : ∀ d e1 e2,
  Pr (e1 ∪ e2) d = Pr e1 d + Pr e2 d - Pr (e1 ∩ e2) d.
Lemma Pr_distributivity : ∀ d e1 e2 e3,
  Pr (e1 ∩ (e2 ∪ e3)) d = Pr ((e1 ∩ e2) ∪ (e1 ∩ e3)) d.
Lemma Pr_cplt: ∀ d e,  Pr ē d = sum d - Pr e d.
...
```

2.3 Transformations of Distributions

Our formalization of distributions also features functions that transform distributions according to operations such as random sampling. Let us consider a concrete example of what these transformations are supposed to achieve. We take stores of variables (as defined at the beginning of this section) for deterministic states. Let us assume that we are given the probabilistic state $(p_0, x = 0), (p_1, x = 1)$ and that we perform a random sampling with probability $0 < p < 1$ of the variable y from the set $\{0, 1\}$. The effect of this random sampling is to multiply the original distribution by the number of possible outcomes of the random sampling (here: $y = 0$ or $y = 1$), each distribution being scaled by the adequate probability (here: p and $1 - p$), as depicted informally in Fig. 1.

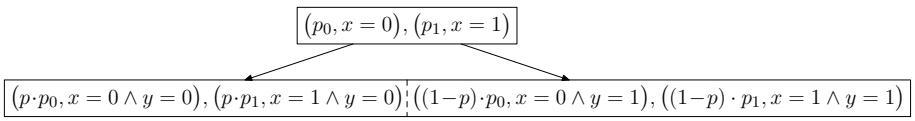

Fig. 1. Effect on a distribution of the random sampling $y \xleftarrow{p} \{0, 1\}$

The function `fork` below implements the most general form of transformation illustrated above. It takes as input a distribution `d` and a list `l` of real scaling factors and functions that transform deterministic states:

```
Fixpoint fork (l : list (R * (A → A))) (d : distrib) : distrib :=
  match l with
    | (k, f) :: tl => map f (scale k d) ++ fork tl d
    | nil => nil
  end.
```

(The function `map` applies a function to each deterministic state of a distribution, the function `scale` multiplies each coefficient by the same real; Coq code omitted to save space.) For example, the transformation depicted in Fig. 1 is performed by the function call `fork ((p, update y 0)::(1-p, update y 1)::nil)` where `update` is a function that updates stores.

3 Formalization of A Probabilistic Language for Games

3.1 Random Oracle

A *random oracle* is a data structure used in security proofs to represent a pseudorandom function or a hash function. Concretely, it is a map from a set of bitstrings to uniformly and independently sampled bitstrings. From a programming language perspective, a random oracle can be thought of as a hash table with random values, as depicted in Fig. 2. Indeed, like a hash table, insertion of new records (key-value pairs) and retrieval of the value associated with a key are

the most important operations. In security proofs, it is also important to be able to look for already allocated keys or values, to talk about the ith inserted record, to know the number of records, etc. The most reusable way to formalize such a rich data structure is via an *abstract datatype*, i.e., a type that is known to enjoy some properties but whose formalization is hidden. In Coq, abstract datatypes are formalized with *modules*. Here follows the type of a module for an abstract datatype `t` that enjoys the properties of a random oracle:

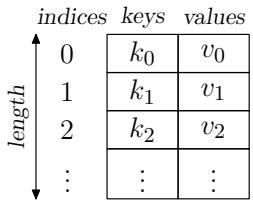

Fig. 2. A simple random oracle

```
Module Type ORACLE.
 Parameter t : Set.
 Parameter empty : t.
 Definition key := nat.
 Definition value := nat.

 (* access functions *)
 Parameter length : t → nat.
 Parameter insert : key → value → t → t.
 Parameter nth_key : nat → t → (*default*)key → key.
 Parameter nth_value : nat → t → (*default*)value → value.
 Parameter find_key : key → t → nat.
 Parameter find_value : value → t → nat.
 ...
 (* properties *)
 Parameter insert_new_len : ∀ o k v,
   find_key k o = 0 → length (insert k v o) = length o + 1.
 ...
End ORACLE.
```

In the following, we use a module `oracle` of type `ORACLE` such that `oracle.t` is the type of our random oracles. Each time we need to manipulate an oracle, we can use functions such as `oracle.length` and we know that they satisfy properties such as `oracle.insert_new_len`. Observe that we use naturals instead of bitstrings but we are careful to keep track of cardinality information; in the future, we plan to use instead the module for machine integers from [14] so that we can handle more faithfully security proofs that make a precise usage of bitstrings (such as the security proof of PSS [5]). For the time being, we have just specified the `oracle` module. A random oracle with one value per key can be implemented on the model of the finite map from [11]; however, most security proofs require random oracles with several values per key, so that we defer to future work the formalization of a module that accommodates such generality.

3.2 Execution State

The random oracle from Sect. 3.1 is only one part of the execution state of games. The other part is the store of variables as we defined in Sect. 2. A deterministic state `dstate` is therefore a pair of a store and an oracle. It is possible to access a

deterministic state to lookup for the value of a variable, or to update the value of a variable (using functions `lookup` and `update` respectively):

```
Definition dstate := store * oracle.t.
Definition lookup (v:var) (d:dstate) : nat := ...
Definition update (v:var) (n:nat) (d:dstate) : dstate := ...
```

Finally, a *probabilistic state* is a distribution (as defined in Sect. 2) of deterministic states: `Definition pstate := distrib dstate.`

3.3 Programming Language

Our probabilistic programming language is an imperative language with random sampling and function calls built after [7].

The expressions `expr` of our programming language contain variables, integer constants, and a C-like negation operator (this can be easily extended but is sufficient for our purpose in this paper):

```
Inductive expr : Set :=
  var_e : var → expr | int_e : nat → expr | neg_e : expr → expr.
```

An expression is a datatype so it belongs to `Set`; since an expression can be made up of other expressions, the `expr` type is introduced by the keyword `Inductive`. Everytime the `Inductive` keyword is used, Coq generates an induction principle for the datatype that allows for proof by induction. An expression `e` can be evaluated in a deterministic state `ds` using the function `eval`:

```
Fixpoint eval (e:expr) (ds:dstate) : nat := ...
```

We now define the commands of our programming language. Random sampling is available through two commands: `x <-$- n` (notation for `sample_n x n`) uniformly samples a natural from the interval $[0, n-1]$, and `x <-b- p` (notation for `sample_b x p`) samples a boolean (in fact, a natural among 0 and 1) with probability $0 < p < 1$. For the random oracle, there are two commands to access it: `insert k v` adds a new record `(k,v)`, `find_value x e` tests for the occurrence of the value `e` and sets the variable `x` accordingly. It is possible to put together atomic commands using control-flow commands: `c1;c2` (notation for `seq c1 c2`) represents a sequence of commands, `ifte b c1 c2` represents an if-then-else with condition `b` (an expression) and two branches `c1` and `c2`. Other commands' names are self-explanatory:

```
Definition fun_id := nat.
Inductive cmd : Set :=
| skip : cmd
| assign : var → expr → cmd          (* Notation: _ <- _ *)
| sample_n : var → nat → cmd         (* Notation: _ <-$- _ *)
| sample_b : var → R → cmd           (* Notation: _ <-b- _ *)
| find_value : var → expr → cmd
| insert : expr → expr → cmd
| ifte : expr → cmd → cmd → cmd
| seq : cmd → cmd → cmd              (* Notation: _ ; _ *)
| call : fun_id → cmd.
```

Like expressions, commands are just another datatype belonging to `Set` and the `Inductive` keyword provides us with an induction principle to reason about the syntax of games.

So far we have only defined the syntax of games without giving them meaning. Now we define their semantics. We do that by formalizing an operational semantics [3] with the logical predicate `_||-_--_-->_` (notation for `(exec____)`) between (1) an environment (a set of function ids and commands), (2) a starting probabilistic state, (3) a command, and (4) a resulting probabilistic state:

```
Definition prog := list (fun_id * cmd).
Inductive exec (prg : prog) : pstate → cmd → pstate → Prop := ...
```

Because this is a logical predicate it belongs to type `Prop`. It is also defined using the `Inductive` keyword so that Coq provides an induction principle to reason by induction on the execution of games; this induction principle is used pervasively in our formal proofs. Let us illustrate the `exec` predicate with the operational semantics of random sampling commands (the complete predicate can be found in Appendix A). The constructor `exec_sample_b` formalizes the semantics of (possibly non-uniform) boolean sampling:

```
| exec_sample_b : ∀ x p st, 0 < p < 1 →
  prg ||- st -- x <-b- p -->
                        fork ((p, update x 1)::(1-p, update x 0)::nil) st
```

Starting from a probabilistic state `st`, the command `x <-b- n` yields a probabilistic state `fork ((p, update x 1)::(1-p, update x 0)::nil) st`. This transformation actually corresponds to the situation depicted in Fig. 1, Sect. 2.3, where the function `fork` has been explained. The constructor `exec_sample_n` formalizes the semantics of uniform random sampling:

```
| exec_sample_n: ∀ x n st, n > 0 →
  prg ||-  st  --  x <-$- n  -->  fork (sample_n_fork_distrib 0 n n x) st
```

Starting from a probabilistic state `st`, the command `x <-$- n` yields a probabilistic state `fork (sample_n_fork_distrib 0 n n x) st`. The function application `sample_n_fork_distrib` produces an appropriate list of scaling factors and state transformations: `(1/n, update x 0)::(1/n, update x 1)::···::(1/n, update x (n-1))`. It is defined by recursion on the size of the sample space:

```
Fixpoint sample_n_fork_distrib
  (min span card:nat) (v:var) : list (R * (dstate -> dstate)) :=
    match span with
      | 0 => nil
      | S span' => (1/INR card, fun x => update v (min + span') x) ::
                     sample_n_fork_distrib min span' card v
    end.
```

(The constructor `S` is the successor function of naturals; the function `INR` injects naturals into reals; the construct `fun x => ...` corresponds to an anonymous function, i.e., it is equivalent to `Definition` *`anonymous_function`* `x := ...`)

Loop Constructs In security proofs, the computational power of the adversary is always bounded. As a consequence, we do not need any general form of while-loops and we can get along with macros for looping constructs. For example, the function below represents i copies of some parameterized loop body:

```
Fixpoint loop (i : nat) (c : nat → cmd) : cmd :=
  match n with
    | O => skip
    | S j => loop j c; c j
  end.
```

3.4 Properties of The Probabilistic Language of Games

Using the probabilistic programming language defined above, we have formally proved in Coq several reusable lemmas to reason about games. The most important of these lemmas is the fundamental lemma of game-playing that is explained in Sect. 4. There are many other useful lemmas. The most important of them are those that capture the properties of random sampling, answering questions such as: "What is the probability that two random variables are equal?" Such a question arises for example when comparing the value stored in a random oracle with a newly sampled value.

For illustration, let us consider the case of two uniformly sampled values. In Coq, lemmas are proved interactively in a special mode entered in via the keyword `Lemma`. The lemma corresponding to the question above consists of two hypotheses. First, we are given an execution step of a game: it goes from state `st` to state `st'` by performing a uniform random sampling whose outcome is stored in variable `x` (this is *`hypo 1`* below). Second, we are given a function `f` that retrieves values from the oracle and we know that in state `st` these values are uniformaly distributed with probability `p` (this is *`hypo 2`* below). The conclusion of the lemma states that the probability that the value of `x` is equal to the value retrieved by `f` is also `p`:

```
Lemma exec_sample_n_twice_Pr : ∀ x n st st' prg,
  (* hypo 1 *) prg ||- st -- x <-$- n --> st' →
  ∀ (f:oracle.t → nat) p,
  (* hypo 2 *) (∀ m, m < n →
    Pr (fun s => beq_nat m (f (get_oracle s))) st = p * sum st) →
  (* conclusion *)
  Pr (fun s => beq_nat (lookup x s) (f (get_oracle s))) st' = p * sum st'
```

(`get_oracle` is a function that extracts the oracle from a deterministic state; `beq_nat` tests natural numbers for equality and returns a Coq boolean.)

4 The Fundamental Lemma of Game-playing

In game-playing, each game represents a sequence of interactions with the adversary, and a security proof consists of a sequence of game transformations. During game-transformation steps, we keep track of the bounds incurred in probability changes in order to derive a bound on the probability that the adversary

wins. This is often achieved by application of the "fundamental lemma of game-playing" [6–8].

4.1 Probabilistic Account

Assume one wants to know the difference `Rabs(Pr e d1 - Pr e d2)` of probabilities of some event `e` in two different probability distributions `d1` and `d2` (`Rabs` is the absolute value in Coq). One way to bound this value is to analyze the event `e` with respect to some other event `f` so that the event `e` can be partitioned into events `e ∩ f` and `e ∩ f̄`. Then one has `Pr e d1 = Pr (e ∩ f) d1 + Pr (e ∩ f̄) d1`, and similarly for `d2`.

The lemma below is a version of the fundamental lemma of game-playing with only distributions (games will be added in the next section). It states that, under the hypothesis `Pr (e ∩ f̄) d1 = Pr (e ∩ f̄) d2` (i), we have a simple bound[4]:

```
Lemma abstract_fundamental_lemma : ∀ d1 d2 e f r, 0 ≤ r →
  sum d1 = sum d2 → coeff_pos d1 → coeff_pos d2 →
    Pr f d1 = Pr f d2 = r → Pr (e ∩ f̄) d1 = Pr (e ∩ f̄) d2 →
      Rabs(Pr e d1 - Pr e d2) ≤ r.
```

The hypotheses `sum d1 = sum d2`, `coeff_pos d1` and `coeff_pos d2` come from how we formalize distributions, see Sect. 2.1. This lemma is proved in Coq as follows: first use the equality (i) to eliminate the terms with `f̄`, then the difference becomes `Rabs(Pr (e ∩ f) d1 - Pr (e ∩ f) d2)`, which is easily seen to be bounded by the probability `r` that `f` occurs.

4.2 Identical-until-bad Games

The lemma of the previous section does not say anything about games, but it can actually be used to transform one game to another if they are "identical-until-bad", a syntactic property that can be automatically verified in Coq for the probabilistic language we introduced earlier.

The property "identical-until-bad" [7] assumes the existence of a special variable conventionally called `bad` that can be set only once. Two games are "identical-until-bad" when they have the same syntax tree except for those subtrees following the command `bad <- 1`. We formalize the property "identical-until-bad" using the logical predicates `no_assign_cmd` and `no_assign` that check for variable assignments; this is not the most general formalization but it is sufficient for our purpose in this paper. For example, assume that we have three commands `c1`, `c2`, `c2'` such that `no_assign_cmd bad` holds, and a program `prg` such that `no_assign bad` holds. Then the following two programs are "identical-until-bad":

```
prg ||- ␣ -- ifte b                 | prg ||- ␣ -- ifte b
      c1                            |       c1
      (bad <- int_e 1; c2) --> ␣    |       (bad <- int_e 1; c2') --> ␣
```

[4] This lemma can actually be generalized to use two pairs of different events `e1`, `e2` and `f1`, `f2` in two different probability distributions `d1` and `d2` and with a bound equal to the maximum of `Pr f1 d1` and `Pr f2 d2`.

The following example shows "identical-until-bad" games containing a loop; here, commands `c1`, `c2`, `c2'`, `c3`, and all `c0 i` for $0 \leq$ `i` $<$ `q` satisfy `no_assign_cmd bad`, and the program `prg` satisfies `no_assign bad`:

```
prg ||- _ -- loop q (fun i =>
        c0 i;
        ifte b
            c1
            (bad <- int_e 1; c2);
        c3) --> _
```

```
prg ||- _ -- loop q (fun i =>
        c0 i;
        ifte b
            c1
            (bad <- int_e 1; c2');
        c3) --> _
```

4.3 Fundamental Lemma of Game-playing : Formal Statement

In this section, we show how to formally state and prove the fundamental lemma of game-playing. For this purpose, we use the lemma of Sect. 4.1 and the property "identical-until-bad" of Sect. 4.2. The relation between the abstract fundamental lemma and the property "identical-until-bad" is the event "the variable `bad` is set to one". We identify the event `e` in Sect. 4.1 with the event that the adversary wins, and the event `f` with the event `sets bad 1` where `sets` is defined as follows:

```
Definition sets (v:var) (n:nat) : event dstate :=
  fun s => beq_nat (lookup v s) n.
```

Keeping this relation in mind, we can now state the fundamental lemma of game-playing. First, we take two "identical-until-bad" games. As seen in Sect. 4.2, this boils down to automatically check some `no_assign bad` and `no_assign_cmd bad` logical predicates. For concreteness, let us consider the two games of the Switching Lemma. Second, we take two initial distributions `st` and `st'` such that:

```
Permutation (filter (sets bad 1) st) (filter (sets bad 1) st')
```
(ii)

With these hypotheses, the following fundamental lemma of game-playing can be proved formally in Coq (`no_assign_cmd_list` is a variant of the `no_assign_cmd` predicate):

```
Lemma fundamental_lemma :
 ∀ prg (c0:nat → cmd) b bad c1 c2 c2' c3 e q st st' end end',
 no_assign_cmd_list bad c1 c2 c2' c3 → (∀ i, no_assign_cmd bad (c0 i)) →
 no_assign bad prg → coeff_pos st → coeff_pos st' → sum st = sum st' →
 Permutation (filter (sets bad 1) st) (filter (sets bad 1) st') →
 prg ||- st  --
   loop q (fun i => c0 i; ifte b c1 (bad <- int_e 1; c2 ); c3) --> end  →
 prg ||- st' --
   loop q (fun i => c0 i; ifte b c1 (bad <- int_e 1; c2'); c3) --> end' →
 Rabs (Pr e end - Pr e end') <= Pr (sets bad 1) end.
```

A detailed account of the proof of this lemma can be found in Appendix B. The next section shows how it can be concretely applied.

5 The PRP/PRF Switching Lemma

The PRP/PRF Switching Lemma is used in security proofs of cryptographic schemes based on block ciphers. Although block ciphers are assumed to behave as pseudorandom permutations (PRP), it is easier to consider them as pseudorandom functions (PRF) in security proofs. The Switching Lemma quantifies in terms of probabilities the difference induced by this approximation. As explained in [7], it is non-trivial to prove this lemma correctly; here follows a formal proof.

5.1 Formal Statement

The proof of the Switching Lemma in game-playing assumes an adversary `A` that does `q` queries to two games `G0` and `G1` that represent respectively a pseudorandom function and a pseudorandom permutation:

```
Definition G0' bad (A:nat→nat) i :=
 x <- int_e (A i) ;
 y <-$- n ;
 find_value z (var_e y) ;
 ifte (var_e z)
   (bad <- int_e 1)
   skip;
 insert (var_e x) (var_e y).

Definition G0 bad q (A:nat→nat) :=
 loop q (G0' bad A).
```

```
Definition G1' bad (A:nat→nat) i :=
 x <- int_e (A i) ;
 y <-$- n ;
 find_value z (var_e y) ;
 ifte (var_e z)
   (bad <- int_e 1; any)
   skip;
 insert (var_e x) (var_e y).

Definition G1 bad q (A:nat→nat) :=
 loop q (G1' bad A).
```

The `bad` variable is set when the function built is not a permutation. The difference between the two games is that, when `bad` is set, `G1` performs a command `any`. This command can be anything that does not modify `bad`; in practice, `any` samples `y` again in a way such that it does build a permutation. We do not need to be specific about what `any` does because it is irrelevant to the proof.

The Switching Lemma is formally stated as follows. Starting with a valid distribution `st` such that `bad` is not set, the execution of games `G0` and `G1` leads to two distributions `st'` and `st''` such that the difference of the probabilities that an event `e` occurs is bounded by $\frac{q(q-1)}{2n}$ (where n is the cardinal of the random sampling):

```
Lemma switching : ∀ q, q ≠ 0 → ∀ A, (∀ x y, x ≠ y → A x ≠ A y) →
  ∀ st, coeff_pos st → sum st > 0 → plength 0 st →
  Pr (sets bad 1) st = 0 →
  ∀ st', nil ||- st -- G0 bad q A --> st' →
  ∀ st'', nil ||- st -- G1 bad q A --> st'' →
  ∀ e, Rabs (Pr e st'' - Pr e st') ≤ INR(q*(q-1))/INR(2*n) * sum st'.
```

5.2 Formal Proof

The proof of the Switching Lemma consists of the successive application of the two following lemmas `switching_part1` and `switching_part2`:

```
Lemma switching_part1 : ∀ q, ∀ A, (∀ x y, x ≠ y → A x ≠ A y) →
  ∀ st, coeff_pos st →
  ∀ st', nil ||- st -- G0 bad q A --> st' →
  ∀ st'', nil ||- st -- G1 bad q A --> st'' →
  ∀ e, Rabs (Pr e st'' - Pr e st') ≤ Pr (sets bad 1) st'.
```

The proof of `switching_part1` is by application of the fundamental lemma of game playing seen in Sect. 4.

The goal of the second part of the Switching Lemma is to upper-bound the probability that the variable `bad` is set in game `G0`. This is a proof by induction on the number of requests of the adversary but it requires a generalization to be handled gracefully. We introduce *probabilistic predicates* for the purpose of generalization. In the same way that events (defined in Sect. 2) are predicates for deterministic states `dstate`, probabilistic predicates are predicates for probabilistic states `pstate`. Technically, a probabilistic predicate is a Coq function of type `pstate → Prop`. For example, the property that all the random oracles of a probabilistic state have the same length is captured by the predicate `plength`:

```
Definition plength (len:nat) (ps:pstate) : Prop :=
  ∀ p ds, (p, ds) ∈ st → oracle.length (get_oracle ds) = len.
```

Similarly, the property that the ith key of all the random oracles of a probabilistic state is k is captured by the predicate `pnth_key` and the property that the ith value of all the random oracles of a probabilistic state is uniformly distributed with probability $\frac{1}{n}$ is captured by the predicate `pnth_value_uniform`:

```
Definition pnth_key (i k:nat) (ps:pstate) :=
  ∀ p ds, (p, ds) ∈ ps → oracle.nth_key' i (get_oracle ds) = Some k.

Definition pnth_value_uniform (i n:nat) (st:pstate) :=
  ∀ m, m < n →
    Pr (fun s => beq_nat m (oracle.nth_value i (get_oracle s) 0)) st =
      1/INR n * sum st.
```

Using probabilistic predicates, the second part of the Switching Lemma is stated as follows:

```
Lemma switching_part2 : ∀ q, q ≠ 0 → ∀ A, (∀ x y, x ≠ y → A x ≠ A y) →
  ∀ st, coeff_pos st → sum st > 0 → plength 0 st →
  Pr (sets bad 1) st = 0 →
  ∀ st', nil ||- st -- G0 bad q A --> st' →
  plength q st' ∧
  (∀ k, k < q → pnth_key k (A k) st' ∧ pnth_value_uniform k n st') ∧
  Pr (sets bad 1) st' ≤ INR(q*(q-1))/INR(2*n) * sum st'.
```

Intuitively, probabilistic predicates `plength`, `pnth_key`, and `pnth_value_uniform` capture how the random oracle is transformed from one loop iteration to the other: at any point of execution, the length of the oracle is equal to the number of queries so far, the keys of the oracle correspond to the queries so far, and associated values are all uniformly distributed. In the following, we briefly skim through the formal proof.

The proof is by induction on the number `q` of adversary queries, like the proof by induction of the Gauss formula for the sum of consecutive integers. In the inductive case, we are led to quantify the difference between the probabilities that `bad` is set before and after an iteration of the loop:

```
Pr (sets bad 1) st' ≤ Pr (sets bad 1) st + INR(q+1)/INR n * sum st'
```

This is equivalent to upper-bounding the probability that the randomly sampled `y` already appears in the values of the random oracle:

```
Pr (fun s => beq_nat (eval (neg_e (var_e z)) s) 0) st
                                                ≤ INR(q+1)/INR n * sum st
```

The probability of the occurence of a value in the values of the random oracle can be upper-bounded by the sum of the probabilities that it is equal to each value, using the general-purpose lemma below:

```
Lemma Pr_iter_orb : ∀ st len e, coeff_pos st → plength len st →
 Pr (fun s => (eval e s) ∈ (oracle.values (get_oracle s))) st ≤
  Sum 0 len (fun x => Pr (fun s =>
    beq_nat (eval e s) (oracle.nth_value x (get_oracle s) 0)) st).
```

(`Sum 0 len f` is a Coq function for $\sum_{x=0}^{len-1} f(x)$.) By the lemma from Sect. 3.4, we know that the probability that the randomly sampled `y` is equal to a (randomly sampled) value of the random oracle is $\frac{1}{n}$:

```
∀ i, i < q+1 → Pr (fun s =>
  beq_nat (lookup y s) (oracle.nth_value i (get_oracle s) 0)) st =
    1/INR n * sum st
```

Because this probability is a constant, the sum inherited from the previous lemma is equal to `INR(q+1)/INR n * sum st`. Using the inductive hypothesis, this completes the proof of the PRP/PRF Switching Lemma.

6 Related Work

CryptoVerif is a tool to automate proofs of cryptographic protocols in the computational model. In particular, it has been applied to security proofs written as sequences of games [13]. In CryptoVerif, games are written in a process calculus and game transformations are captured by probabilistic bisimulation relations. The validity of game transformations sometimes require non-trivial manual proofs (see Appendix B of [13]). Though our formalization of the game-playing framework is not as rich as CryptoVerif, our Coq-centric approach provides a way to avoid manual proofs.

Our probabilistic programming language with probabilities and probabilistic predicates is reminiscent of probabilistic Hoare logic. The latter has been used to build the IND-CPA security proof of the ElGamal encryption scheme [12]. Though manual, this proof is so detailed that we think it is close to being formalized. Our formalization of the game-playing framework is not strictly speaking a formalization of probabilistic Hoare logic, but it gives a good idea of the effort

it would require and, more importantly, how to extend it with random oracles (which is not done in [12]).

There exist other formalizations of security proofs in the Coq proof assistant. An early work makes use of the Generic Model and of the Random Oracle Model but without game-playing [10]. In this formalization, an adversary is defined as an inductive set of possible actions and this allows for reasoning about its chances to resolve a challenge. Yet, no use-case has been completely formalized that demonstrates the effectiveness of this approach. A recent work formalizes the IND-CPA security proof of the ElGamal encryption scheme by game-playing in the standard model [15]. In this formalization, games are encoded directly as Coq functions; the absence of syntax seems to simplify formal reasoning but it is likely to hinder automation of game transformations, which are syntactic in nature. Besides this issue, we regard this work as complementary to ours: it provides an IND-CPA security proof using a formalization of cyclic groups that can be easily integrated in our formalization.

7 Conclusion

In this paper, we explained a formalization of the game-playing framework of Bellare and Rogaway in the Coq proof assistant. Our formalization features a probabilistic language to write games and several reusable lemmas to carry out security proofs, including in particular an instance of the fundamental lemma of game-playing. We have illustrated the usefulness of our formalization by proving the PRP/PRF Switching Lemma. The complete Coq development is available online [16]. To our knowledge this is the first formalization of game-playing with a random oracle and a working fundamental lemma used in a complete use-case.

Future Work For the time being, the fundamental lemma of game-playing as it appears in Sect. 4 can only be applied to a restricted set of games. Of course, we can easily formalize variants on the same model, but ideally it should be generalized so as to encompass any pair of "identical-until-bad" games [7].

We already have a good idea of how to extend our formalization of the game-playing framework to carry out the security proof of the Full-domain Hash signature scheme from [8]. This will require a formalization of random oracles with several values per key and, more importantly, introduce concurrency issues arising from the parallel execution of several oracles.

Acknowledgments The first and second authors acknowledge partial support from, respectively, the Grant-in-Aid of Special Coordination Funds for Promoting Science and Technology, and the Grant-in-Aid for Young Scientists (B) (Grant number 187602935003), both by the Ministry of Education, Culture, Sports, Science and Technology, Japan (MEXT).

References

1. The LogiCal Project, INRIA. The Coq proof assistant: `http://coq.inria.fr`.
2. Thompson, S. *Type Theory and Functional Programming*. Addison-Wesley, 1991.
3. Winskel, G. *The Formal Semantics of Programming Languages*. MIT Press, 1993.
4. Mihir Bellare and Phillip Rogaway. Random Oracle are Practical: A Paradigm for Designing Efficient Protocols. In *1st ACM Conference on Computer and Communications Security (CCS 1993)*, p. 62–73. ACM Press.
5. Mihir Bellare and Phillip Rogaway. The Exact Security of Digital Signatures—How to Sign with RSA and Rabin. In *Advances in Cryptology (EuroCrypt 1996)*, volume 1070 of *Lecture Notes in Computer Science*, p. 399–416. Springer.
6. Victor Shoup. Sequence of Games: A Tool for Taming Complexity in Security Proofs. Manuscript. Available at `http://www.shoup.net/papers/games.pdf`. 2004. Revised 2006.
7. Mihir Bellare and Phillip Rogaway. Code-Based Game-Playing Proofs and the Security of Triple Encryption. In *Advances in Cryptology (EuroCrypt 2006)*, volume 4004 of *Lecture Notes in Computer Science*, p. 409–426. Springer. Extended version: Cryptology ePrint Archive: Report 2004/331.
8. David Pointcheval. Provable Security for Public Key Schemes. In *Contemporary Cryptology, Advanced Courses in Mathematics CRM Barcelona*, p. 133–189. Birkhäuser Publishers, 2005.
9. Shai Halevi. A plausible approach to computer-aided cryptographic proofs. Cryptology ePrint Archive: Report 2005/181.
10. Sabrina Tarento. Machine-Checked Security Proofs of Cryptographic Signature Schemes. In *10th European Symposium on Research in Computer Security (ESORICS 2005)*, volume 3679 of *Lecture Notes in Computer Science*, p. 140–158. Springer.
11. Nicolas Marti, Reynald Affeldt, and Akinori Yonezawa. Formal Verification of the Heap Manager of an Operating System using Separation Logic. In *8th International Conference on Formal Engineering Methods (ICFEM 2006)*, volume 4260 of *Lecture Notes in Computer Science*, p. 400–419. Springer.
12. Ricardo Corin and Jerry den Hartog. A Probabilistic Hoare-style Logic for Game-Based Cryptographic Proofs. In *33rd International Colloquium on Automata, Languages and Programming (ICALP 2006)*, volume 4052 of *Lecture Notes in Computer Science*, p. 252–263. Springer.
13. Bruno Blanchet and David Pointcheval. Automated Security Proofs with Sequences of Games. In *26th Annual International Cryptology Conference (CRYPTO 2006)*, volume 4117 of *Lecture Notes in Computer Science*, p. 537-554. Springer. Extended version: Cryptology ePrint Archive: Report 2006/069.
14. Reynald Affeldt and Nicolas Marti. An Approach to Formal Verification of Arithmetic Functions in Assembly. In *11th Annual Asian Computing Science Conference (ASIAN 2006), Focusing on Secure Software and Related Issues*, to appear in *Lecture Notes in Computer Science*. Springer.
15. David Nowak. A Framework for Game-Based Security Proofs. Cryptology ePrint Archive: Report 2007/199.
16. Reynald Affeldt, Miki Tanaka, and Nicolas Marti. Formal Proof of Provable Security by Game-playing in a Proof Assistant. Coq scripts. Available at `http://staff.aist.go.jp/reynald.affeldt/secprf/provsec2007`.

A Formalization Excerpt: Operational Semantics

```
Inductive exec (prg : prog) : pstate → cmd → pstate → Prop :=
| exec_skip : ∀ st, prg ||- st -- skip --> st

| exec_assign : ∀ x e st,
  prg ||- st -- x <- e -->
    fork ((1, fun s => update x (eval e s) s)::nil) st

| exec_sample_b : ∀ x p st, 0 < p < 1 →
  prg ||- st -- x <-b- p -->
    fork ((p, update x 1)::(1-p, update x 0)::nil ) st

| exec_find_value : ∀ x st e,
  prg ||- st -- find_value x e -->
    fork ((1, fun s =>
      update x (oracle.find_value (eval e s) (get_oracle s)) s)::nil) st

| exec_sample_n: ∀ x n st, n > 0 →
  prg ||- st -- x <-$- n --> fork (sample_n_fork_distrib 0 n n x) st

| exec_ifte : ∀ e c d st st_true st_false stc std,
  st_true = filter (fun s => beq_nat (eval (neg_e e) s) 0) st →
    st_false = filter (fun s => beq_nat (eval e s) 0) st →
      prg ||- st_true -- c --> stc →
        prg ||- st_false -- d --> std →
          prg ||- st -- ifte e c d --> stc ++ std

| exec_seq : ∀ st st'' st' c d,
  prg ||- st -- c --> st'' →
    prg ||- st'' -- d --> st' →
      prg ||- st -- c ; d --> st'

| exec_insert : ∀ st e e',
  prg ||- st -- insert e e' -->
    fork ((1, fun s => (get_store s,
      oracle.insert (eval e s) (eval e' s) (get_oracle s)))::nil) st

| exec_call : ∀ st st' callee c,
  get_fun_cmd prg callee = Some c →
    prg ||- st -- c --> st' →
      prg ||- st -- call callee --> st'
where "prg ||- st -- c --> st'" := (exec prg st c st').
```

B Proof Sketch for the Lemma of Sect. 4.3

In order to prove the fundamental lemma of game-playing, we need the following two lemmas `identical_until_bad` and `after_bad_is_set` that relate the condition (ii) and the executions of "identical-until-bad" commands. The lemma `identical_until_bad` states that, given two games that are "identical-until-bad" (defined with `no_assign_cmd bad` and `no_assign bad` predicates), the condition (ii) is preserved by the execution of the games:

```
Lemma identical_until_bad :
 ∀ prg (c0:nat → cmd) b bad c1 c2 c2' c3 q st st' end end',
 no_assign_cmd_list bad c1 c2 c2' c3 → (∀ i, no_assign_cmd bad (c0 i)) →
 no_assign bad prg → coeff_pos st → coeff_pos st' →
 Permutation (filter (¬sets bad 1) st) (filter (¬sets bad 1) st') →
 prg ||- st --
   loop q (fun i => c0 i; ifte b c1 (bad <- int_e 1; c2); c3) --> end →
 prg ||- st' --
   loop q (fun i => c0 i; ifte b c1 (bad <- int_e 1; c2'); c3) --> end' →
 Permutation (filter (¬sets bad 1) end) (filter (¬sets bad 1) end').
```

Concerning the lemma `after_bad_is_set`, note that when both (ii) and `sum st = sum st'` hold, the two initial distributions of the execution have the same probabilities for the `sets bad 1` event. The lemma states that this property is also preserved after the execution of "identical-until-bad" games:

```
Lemma after_bad_is_set :
 ∀ prg (c0:nat → cmd) b bad c1 c2 c2' c3 q st st' end end',
 no_assign_cmd_list bad c1 c2 c2' c3 → (∀ i, no_assign_cmd bad (c0 i)) →
 no_assign bad prg → coeff_pos st → coeff_pos st' → sum st = sum st' →
 Permutation (filter (¬sets bad 1) st) (filter (¬sets bad 1) st') →
 prg ||- st --
   loop q (fun i => c0 i; ifte b c1 (bad <- int_e 1; c2); c3) --> end →
 prg ||- st' --
   loop q (fun i => c0 i; ifte b c1 (bad <- int_e 1; c2'); c3) --> end' →
 Pr (sets bad 1) end = Pr (sets bad 1) end'.
```

The proof of the fundamental lemma of game-playing proceeds along the lines of the proof of the abstract fundamental lemma of Sect. 4.1. First, we prove `Pr (sets bad 1) end = Pr (sets bad 1) end'` by direct application of the lemma `after_bad_is_set`. Second, using the lemma `identical_until_bad` we prove

```
Pr (f ∩ (¬sets bad 1)) end = Pr (f ∩ (¬sets bad 1)) end'
```

which is equivalent to the condition (i) in Sect. 4.1. We use this equality to eliminate the terms with complement. Then the difference becomes

```
Rabs (Pr (f ∩ (sets bad 1)) end - Pr (f ∩ (sets bad 1)) end')
```

and the rest of the argument is exactly the same as the case of abstract fundamental lemma.