Reflection, Rewinding, and Coin-Toss in EasyCrypt

Denis Firsov
Guardtime
Tallinn University of Technology
Tallinn, Estonia
denis.firsov@guardtime.com

Dominique Unruh
Tartu University
Tartu, Estonia
unruh@ut.ee

***Abstract*—In this paper we derive a suit of lemmas which allows users to internally reflect EasyCrypt programs into distributions which correspond to their denotational semantics (probabilistic reflection). Based on this we develop techniques for reasoning about rewinding of adversaries in EasyCrypt. (A widely used technique in cryptology.) We use reflection and rewindability results to prove the security of a coin-toss protocol.**

***Index Terms*—cryptography, formal methods, EasyCrypt, reflection, rewindability, commitments, binding, coin-toss**

Contents

1. Introduction

Handwritten cryptographic security proofs are inherently error-prone. Humans will make mistakes both when writing and when checking the proofs. To ensure high confidence in cryptographic systems, we use frameworks for computer-aided verification of cryptographic proofs. One widely used such framework is the EasyCrypt tool [1]. In EasyCrypt, a cryptographic proof is represented by a sequence of "games" (simple probabilistic programs), and the relationship between programs are analyzed in a probabilistic relational Hoare logic (pRHL). EasyCrypt has been successfully used to verify a variety of cryptographic schemes, e.g., (electronic voting [2], digital signatures [3], differential privacy [4], security of IPsec [5], and many others).

However, there is one cryptographic proof technique that seems very difficult to implement in EasyCrypt, namely rewinding. Rewinding is ubiquitous in more advanced proofs, especially when involving zero-knowledge proofs, multi-party computation, but also in relatively simple cases such as building a coin-toss from a commitment. (The latter case we will explore in Sec. 5.) In a nutshell, rewinding refers to the proof technique in which we take a given (usually unknown) program A (an adversary), and convert it into an adversary B that performs the following or similar steps:

1) Remember the initial state of A.
2) Run A.
3) Restore the original initial state of A.
4) Run A again.
5) Combine the results from the runs and/or repeat this until it yields a desired outcome.

While the above steps seem simple, we run into numerous challenges when trying to implement rewinding in EasyCrypt, both due to restrictions in the type system, and due to the necessity for reasoning about probability distributions of program outputs in a way that is not directly supported by EasyCrypt's tactics.

To the best of our knowledge, rewinding has not been implemented in EasyCrypt, nor in other frameworks for reasoning about cryptographic proofs such as CryptHOL [6] in Isabelle, FCF [7] in Coq, CryptoVerif [8] in OCaml, Verypto [9] in Isabelle, CertiCrypt [10] in Coq.

Our contribution. In this work, we design a set of tools to address rewindability in the EasyCrypt framework, and for reasoning about the probabilistic semantics of programs inside EasyCrypt (probabilistic reflection). We validate our results by developing a formal proof of a coin-toss protocol based on rewinding. This paper is accompanied by EasyCrypt code which can be found here [11].

1.1. Challenges

To understand the motivation behind our project, let us look at the example of a pen-and-paper derivation using rewinding. When analyzing a coin-toss protocol based on a commitment, we are faced with the following situation: We have an adversary A that can open the commitment to contain b given input $b = \mathit{false}, \mathit{true}$ (with some non-zero probability). We want to show that this means that A could also, in the same run, produce openings to both *false* and *true*. This is done by defining a different adversary B that runs *A.commit* (to produce the commitment), stores the state of A, runs *A.open(false)* (to produce the first opening), restores the state of A, and runs *A.open(true)* (to produce the second opening). Then we show that the probability that B produces two valid openings is lower-bounded in terms of the probability that A is successful in producing one valid opening. As we will see in more detail in Sec. 5.1, the core theorem for showing this is the following.

Theorem 1.1. Let A be a probabilistic program and let $\mathbf{m}$ denote a memory configuration which represents an initial state of A. We write $\Pr[\, r \leftarrow X.p() \;@\; \mathbf{m} : M \,]$ to denote the probability of a predicate M satisfying the result of running the procedure *p()* of a module X on the initial memory $\mathbf{m}$. In this case, the following inequality holds:

$$\Pr\left[\begin{array}{c} A.init();\; s \leftarrow A.getState(); \\ r_1 \leftarrow A.main();\; A.setState(s); \\ r_2 \leftarrow A.main() \;@\; \mathbf{m} : r_1 \wedge r_2 \end{array}\right] \geq \Pr[\, A.init();\; r \leftarrow A.main() \;@\; \mathbf{m} : r \,]^2 .$$

Proof: Step (1) applies "the averaging technique" by representing *A.init()* as a family of distributions $D^{\mathbf{m}}_A$. We write $\mu_1(D^{\mathbf{m}}_A, \mathbf{n})$ for the probability that $D^{\mathbf{m}}_A$ assigns to $\mathbf{n}$. Then $\mu_1(D^{\mathbf{m}}_A, \mathbf{n})$ is the probability of $A.init()$ terminating in the memory state $\mathbf{n}$ given that it starts in the initial state $\mathbf{m}$. The rest of the computations are then run starting from memory $\mathbf{n}$.

$$\begin{aligned} &\Pr\left[\begin{array}{c} A.init();\; s \leftarrow A.getState(); \\ r_1 \leftarrow A.main();\; A.setState(s); \\ r_2 \leftarrow A.main() \;@\; \mathbf{m} : r_1 \wedge r_2 \end{array}\right] \\ &\overset{(1)}{=} \sum_{\mathbf{n}} \mu_1(D^{\mathbf{m}}_A, \mathbf{n}) \cdot \Pr\left[\begin{array}{c} s \leftarrow A.getState(); \\ r_1 \leftarrow A.main(); A.setState(s); \\ r_2 \leftarrow A.main() \;@\; \mathbf{n} : r_1 \wedge r_2 \end{array}\right] \\ &\overset{(2)}{=} \sum_{\mathbf{n}} \mu_1(D^{\mathbf{m}}_A, \mathbf{n}) \cdot \Pr[r \leftarrow A.main() \;@\; \mathbf{n} : r]^2 \\ &\overset{(3)}{\geq} \left(\sum_{\mathbf{n}} \mu_1(D^{\mathbf{m}}_A, \mathbf{n}) \cdot \Pr[r \leftarrow A.main() \;@\; \mathbf{n} : r]\right)^2 \\ &\overset{(4)}{=} \Pr[\, A.init();\; r \leftarrow A.main() \;@\; \mathbf{m} : r \,]^2 . \end{aligned}$$

Step (2) makes use of the fact that the probability of a success (i.e., *A.main* returning *true*) in both of two independent runs equals to the square of a probability of a success in a single run. Step (3) is an application of Jensen's inequality. The final step undoes the averaging. □

There are number of challenges in performing this proof formally in EasyCrypt:

1) Our proof turns the program *A.init()* into a parameterized distribution of final memories (memories after *A.init()* terminates). While EasyCrypt has a notion $\Pr[\, A.init() \;@\; \mathbf{m} : M \,]$ it does not formally recognize that this defines a distribution, not withstanding the suggestive syntax.
2) Also, our proof makes use of results about probability distributions (e.g., Jensen's inequality) which can be easily stated and proved in terms of probabilistic distributions while they would be much harder to express and prove directly using $\Pr[\ldots]$ expressions or similar program logic constructs and tactics.
3) Another challenge is that EasyCrypt does not have a type "memory"; we cannot define a distribution over memories because we cannot even assign it a type. In EasyCrypt, memories are recognized purely syntactically by prepending the variable names with &.
4) Finally, it is not immediate how one can generically specify the interface of programs (modules) which can return their own state. In other words, what must be the type of the return value of the function *A.getState()*?

1.2. Probabilistic Reflection

As explained above one of the challenges is that we cannot access the distribution which corresponds to the semantics of the program. To enable this, we introduce a suit of lemmas which allows us to access the distribution corresponding to a program.

This turned out to be a powerful tool for rewinding proofs, but we also believe that it can be useful when one needs to derive facts for programs based on their denotational semantics. For example, in a situation when a particular tactic is not available in EasyCrypt, but in a pen-and-paper proof one would show it simply based on probability theory reasoning (e.g., Jensen's inequality, averaging). In the following, we sketch our solution for probabilistic reflection.

Recall that there are no valid types which refer to distribution of final memories. These would be needed to give a type to the denotational semantics of a program, let alone define those semantics. However, in EasyCrypt each program has an associated variable $\mathcal{G}_A^{\mathbf{m}}$ (the type of $\mathcal{G}_A^{\mathbf{m}}$ is $\mathcal{G}_A$) which refers to the part of the memory $\mathbf{m}$ accessible by module A. It is guaranteed that running a program A will never change anything outside $\mathcal{G}_A^{\mathbf{m}}$. So "effectively", the semantics of a program can be described by looking only at the $\mathcal{G}_A$-part of a memory. So, we define a family of distributions D_A^g for g of type $\mathcal{G}_A$ such that $\mu_1(D_A^g, h)$ is the probability that we get h in $\mathcal{G}_A$-part of the final memory configuration when starting with g in $\mathcal{G}_A$-part of the initial memory configuration.

Another problem is that we cannot refer to the type $\mathcal{G}_A$ in the top-level definitions (global definitions of operators/constants). So, in particular, we cannot define a distribution D_A^g parameterized by $\mathcal{G}_A$-values in EasyCrypt. In our workaround to this problem, we prove lemmas of the existence of that family of distributions, but we do not define a constant referring to that family. This works since we only need to refer to the type of D_A^g locally in the theorem statement. Then, when reasoning, one can inside a proof refer to "the" distribution that exists by our lemmas.

In conclusion, our lemma for probabilistic reflection looks roughly as follows:

Theorem 1.2. For all memories $\mathbf{m}$ and programs A there exists a family of distributions D_A^g (with g of type $\mathcal{G}_A$) such that for all predicates M on values of type $\mathcal{G}_A$:

$$\mathsf{Pr}\left[\,A.\mathit{main()}\ @\ \mathbf{m} : M(\mathcal{G}_A^{\mathbf{fin}})\,\right] = \mu(D_A^{\mathcal{G}_A^{\mathbf{m}}}, M).$$

Here, **fin** is the final memory after execution of A.*main()* and $\mu(d, M)$ denotes the probability that the predicate M holds for values distributed according to d. Actual reflection lemma adds generality, e.g., referring also to the inputs/outputs of A.*main*, see Sec. 3.1.

However, being able to reflect the distribution corresponding to a given program is not enough. If we want to reason about composite programs, we will also need to understand how the different constructs in our language operate on the distributions. For example, given a program $A; B$, the distribution D_{AB} of the final state can be expressed in terms of the distributions D_A, D_B corresponding to A and B (monadic bind). This does not follow merely from the existence of the reflected distribution for $A; B$ since it would hypothetically be possible for $A; B$ to have a semantics completely unrelated to those of A and B individually. Thus we prove additional lemmas for this and other cases that allow us to derive the distribution of a more complex program from the distribution of its components (see Sec. 3.3).

Altogether this gives us a library for probabilistic reflection in EasyCrypt, independent of the results on rewinding below. See Sec. 3 for details.

1.3. Rewinding

The final challenge in the formal derivation of Thm. 1.1 is that in EasyCrypt, we cannot define a generic interface of modules which return their own state. Morally, we want A.*getState()* to return a value $\mathcal{G}_A^{\mathbf{m}}$ of type $\mathcal{G}_A$. However, this is impossible since the type $\mathcal{G}_A$ is only allowed to appear in the logical statements and program code of other modules but not in the code of the module A itself.

We solve the above problem by defining what it means for a module to be rewindable. In essence, a module is rewindable if and only if the state of the program can be encoded as a bitstring (or equivalently, as any other countable type). In particular, a program with variables of type "real" (which is uncountable) would not be rewindable in that sense. A security proof using rewinding would then only apply to rewindable adversaries which is not a restriction from the cryptographic point of view. (Typically, cryptographic adversaries are assumed to operate on data that is representable in a computer. Such data can always be encoded as a bitstring.)

In conclusion, our definition for rewindable modules (programs) roughly requires a module to have procedures *getState* and *setState*. The execution of A.*getState()* in state $\mathbf{m}$ must return the value $f(\mathcal{G}_A^{\mathbf{m}})$ where f is an arbitrary injective mapping from the type $\mathcal{G}_A$ to some parameter type *sbits*. The *setState* procedure gets an argument x : *sbits* and sets $\mathcal{G}_A^{\mathbf{m}}$ to $f^{-1}(x)$ if $f^{-1}(x)$ is defined.

Altogether this gives an approach for working with rewindable adversaries in EasyCrypt. See Sec. 4 for details.

2. Preliminaries

In this section we review the syntax and semantics of the main EasyCrypt constructs. Readers familiar with EasyCrypt can skip this section and just familiarize themselves with our syntactic conventions in this footnote[1]. The top-level definitions in EasyCrypt consist of types, operators, lemmas/axioms, module types, and modules. In EasyCrypt one can specify datatypes and operators, where types intuitively denote non-empty sets of values and operators are typed pure functions on these sets. EasyCrypt provides basic built-in types such as `unit`, `bool`, `int`, etc. The standard library includes formalizations of lists, arrays, finite sets, maps, probability distributions, etc. EasyCrypt also allows users to implement their own datatypes and functions (including inductive datatypes and functions defined by pattern matching). For example, we can give a definition of a polymorphic identity function as follows:

```
op id ['a] : 'a → 'a = λ x. x.
```

In this paper, for ease of readability, we use a more compact notation `λ x. x` for lambda-abstractions. In the original EasyCrypt code, this would be written as `fun x => x`.

The ambient logic in EasyCrypt is based on a classical (i.e., non-constructive) set theory which we can use to

1. We write $\leftarrow$ for both `<-` and `<@`, $\overset{\$}{\leftarrow}$ for `<$`, $\wedge$ for `/\`, $\vee$ for `\/`, $\leq$ for `<=`, $\forall$ for `forall`, $\exists$ for `exists`, $\mathbf{m}$ for `&m`, $\mathcal{G}_A$ for `glob A`, $\mathcal{G}_A^{\mathbf{m}}$ for `(glob A){m}`, $\lambda x.\ x$ for `fun x => x`, $\times$ for `*`, `t` $\mathcal{L}$ for `t list`, `t` $\mathcal{D}$ for `t distr`, μ for `mu`, μ_1 for `mu1`, `t` $\mathcal{O}$ for `t option`. Furthermore, in `Pr`-expressions, in abuse of notation, we allow sequences of statements instead of a single procedure call. It is to be understood that this is shorthand for defining an auxiliary wrapper procedure containing those statements.

state and prove properties. (The term ambient logic refers to a built-in logic in EasyCrypt. Ambient logic is not specific to reasoning about programs). For example, we can prove that application of `id` to any `x` equals to `x`. In lemmas and axioms we will use symbols ∀ and ∃ instead of the EasyCrypt syntax which uses keywords `forall` and `exists`, respectively.

```
lemma id_prop ['a] : ∀ (x : 'a), id x = x.
proof. trivial. qed.
```

In EasyCrypt, the proof starts with keyword `proof`. The steps of the proof consist of tactic applications (e.g., `auto`, `trivial`, etc.) which either discharge the proof obligation or transform it into subgoal(s). The `qed` finishes the proof.

Types and operators without definitions are abstract and can be seen as parameters to the rest of the development. Parameters can additionally be restricted by axioms. For example, we can parameterize the development by a uniform distribution of elements of type `bits`.

```
type bits.

op bD : bits 𝒟.
axiom bDU : is_uniform bD.
```

The theory containing this axiom can later be "cloned" and the operator `bD` instantiated with a value for which the axiom `bDU` is actually provable. (This enables modular design of theories.)

In this paper we will use notation `bits` $\mathcal{D}$ as a more concise version of the EasyCrypt syntax `bits distr` which denotes the type of distributions of `bits`. Similarly, we will use `bits` $\mathcal{L}$ instead of `bits list`, and `bits` $\mathcal{O}$ instead of `bits option`.

Modules. In EasyCrypt, modules consist of typed global variables and procedures. The set of all global variables of a module is the union of the set of global variables that are declared in that module and the set of all global variables (declared in other modules) which the module could read or write by a series of procedure calls beginning with a call of one of its procedures. In EasyCrypt, the whole memory (state) of a program is referred to by `&m` (or `&n` etc.). We can refer to the tuple of all global variables of the module `A` in `&m` as `(glob A){m}`. The type of all global variables of `A` (i.e., the type of `(glob A){m}`) is denoted by `glob A`. For readability, we will use syntax $\mathcal{G}_A$ for the type `glob A`. Memories `&m` will be typed in bold without the `&` (i.e., **m** for `&m`). And $\mathcal{G}_A^{\mathbf{m}}$ will denote the EasyCrypt value `(glob A){m}`.

For illustration, we implement the following example of a guessing-game module `GG`:

```
module GG = {
  var win : bool
  var c, q : int

  proc init(q : int) = {
    c    ← 0;
    win  ← false;
    GG.q ← q;
  }

  proc guess(x : bits) : bool = {
    var r;
    if (c < q){
      r $← bD;
      win ← win || r = x;
      c ← c + 1;
    }
    return win;
  }
}.
```

The module `GG` has three global variables: `c` and `q` of type `int`, and `win` of type `bool`. Hence, for any memory **m**, $\mathcal{G}_{GG}^{\mathbf{m}}$ has type $\mathcal{G}_{GG}$ which equals to a product `bool × int × int`. The `GG` module allows a player to guess (call the `GG.guess` procedure) a next value sampled from distribution `bD`. The player has at most `q` attempts (set during initialization by procedure `GG.init`). The player wins if they guess correctly at least once.

Module types. In EasyCrypt, module types specify the types of a set of module procedures [12]. Therefore, module types in EasyCrypt are similar to interfaces in other programming languages (e.g., Java). We can specify the module type of `GG` as follows:

```
module type GuessGame = {
  proc init() : unit
  proc guess(x : bits) : bool
}.
```

Note that module types say nothing about the global variables a module could have and only specify the input and output types of the module procedures.

Next, we define a module type of protocol parties (adversaries), who receive an instance `G` of a guessing game as a module parameter. An adversary must have a `play` procedure which starts the game:

```
module type Adversary(G : GuessGame) = {
  proc play() : unit {G.guess}
}.
```

To forbid adversaries to reinitialize the game the `play` procedure can only execute the `guess` procedure of the parameter game `G`. This is optionally expressed by listing the allowed method(s) in the curly braces next to the procedure.

Probability expressions. EasyCrypt has built-in `Pr`-expressions which can be used to refer to the probabilities of events in program executions: `Pr[r ← X.p() @ m : M r]` denotes the probability that the return value `r` of procedure `p` of module `X` given initial memory **m** satisfies the predicate `M`. (I.e., the general form is `Pr[program @ initial memory : event]`.) The `Pr`-notation in EasyCrypt is somewhat restrictive, the program can only be a single procedure call. In our presentation, we relax this notation and allow multiple statements; it is to be understood that in the actual EasyCrypt code this is implemented by defining an auxiliary wrapper procedure that contains those statements.

For example, we can express that for any adversary `A` the probability of winning the guessing-game is smaller or equal than $\frac{q}{n}$, where n is the size of the support of distribution used by `GG` and q is the maximal allowed number of guesses.

```
lemma winPr m : ∀ (A <:Adversary {GG}) q, 0 ≤ q
 ⇒ Pr[GG.init(q); A(GG).play() @ m : GG.win]
    ≤ q / (supp_size bD).
```

(In EasyCrypt, `X :> T` states that the module `X` satisfies the module type `T`.) Note that the module type `Adversary` also includes adversaries who simply set the value `GG.win` to `true`. Luckily, EasyCrypt allows us to write `Adversary{GG}` to denote a subset of adversaries who has disjoint set of global variables from the module `GG`.

3. Toolkit for Probabilistic Reflection

In this section, we discuss a derivation of probabilistic reflection for programs (i.e., modules) in EasyCrypt. Recall that by probabilistic reflection, we mean tools to get access to probabilistic denotational semantics of imperative programs inside EasyCrypt proofs. (Without needing any meta-reasoning.) Also, we use the probabilistic reflection to derive a powerful toolkit of lemmas which is common to pen-and-paper proofs when arguing about distributions underlying programs.

3.1. Probabilistic Reflection

Recall that in Sec. 1.2, we introduced Thm. 1.2 that proves the existence of a distribution corresponding to a program's denotational semantics. In EasyCrypt, we formally state this theorem as follows:

```
lemma reflection_simple : ∃ (D : $\mathcal{G}_A \to \mathcal{G}_A$ $\mathcal{D}$),
  ∀ m M, μ (D $\mathcal{G}_A^{\mathbf{m}}$) M = Pr[A.main() @ m : M $\mathcal{G}_A^{\mathtt{fin}}$].
```

Here, `D g` corresponds to the family D_A^g from Thm. 1.2, i.e., `D` $\mathcal{G}_A^{\mathbf{m}}$ is the supposed distribution of final states of the program after running on initial memory $\mathbf{m}$.

Inside an EasyCrypt proof, this lemma could be used as `elim (reflection_simple A) => D H_D`; this will introduce a variable `D` in the environment of the proof, together with its defining property `H_D` stating the relationship between `D` and `Pr[A.main()...]`.

However, `reflection_simple` as stated is not general enough for many purposes. In particular, if `A.main` takes an argument `a` or returns a value `r` then we cannot reason about the distribution of `r` and express how `D` depends on `r`. The following more general reflection lemma removes these limitations:

```
lemma reflection :
  ∃ (D : $\mathcal{G}_A$ → at → (rt × $\mathcal{G}_A$) $\mathcal{D}$),
  ∀ m M a, μ (D $\mathcal{G}_A^{\mathbf{m}}$ a) M
    = Pr[r ← A.main(a) @ m : M (r, $\mathcal{G}_A^{\mathtt{fin}}$)].
```

The intuition behind this lemma and the previous one is the same. The only difference is that `D` now has an additional argument `a`, referring to the input of `A.main`, and the resulting distribution `(D` $\mathcal{G}_A^{\mathbf{m}}$ `a)` is a distribution over pairs `(r,` $\mathcal{G}_A^{\mathtt{fin}}$`)` of output and final memory.

Note that while EasyCrypt allows us to all-quantify over the module `A` and over the argument and input types (i.e., `at` and `rt`), we cannot quantify over the name `main` of the procedure. Similarly, the number of arguments of procedure `main` is fixed in the lemma. Fortunately, this only constitutes a minor inconvenience, not a real restriction because we can always define a wrapper module `A'` that has a procedure `main` with a single argument `a` (possibly of a tuple type). Then `main` can untuple `a` and invoke the procedure that we actually want to investigate. A simple call to the tactic `inline A'.main` in the proof will then unwrap this wrapper procedure.

In EasyCrypt, we directly prove `reflection` and derive `reflection_simple` as an immediate corollary. For readability, here we describe a direct proof of `reflection_simple` instead.

Proof: We start the proof by defining a predicate `P` on probabilities which is parameterized by an initial state `g` of type $\mathcal{G}_A$ and an element `x` of type $\mathcal{G}_A$. Below we use the EasyCrypt tactic `pose` to give a definition which is local to the proof.

```
pose P g x := λ p. ∀ n,
  $\mathcal{G}_A^{\mathbf{n}}$ = g ⇒ Pr[r ← A.main() @ n : $\mathcal{G}_A^{\mathtt{fin}}$ = x] = p.
```

The probability `p` satisfies the predicate `(P g x)` if it equals to the probability of `A.main` terminating in the state `x` by starting its run from a memory $\mathbf{n}$ which has $\mathcal{G}_A$ variables equal to `g`.

Before continuing with the proof, we explain that the standard library of EasyCrypt provides the formalization of the Axiom of Choice in the form of the operator `choiceb` and its corresponding property `choicebP`:

```
op choiceb ['a] : ('a → bool) → 'a.

axiom choicebP ['a] : ∀ (P : 'a → bool),
  (∃ (x : 'a), P x) ⇒ P (choiceb P).
```

It states that for any predicate `P` if there exists an element which satisfies it then the element denoted by `(choiceb P)` satisfies `P`. Here, it is worth mentioning that all propositions in EasyCrypt have type `bool`.

The next step of the proof is to define a function `(Q g x)` which uses the choice operator on the predicate `(P g x)` to assign a probability to `x`.

```
pose Q g x := choiceb (P g x).
```

Intuitively, `(Q g x)` returns "the" probability `Pr[A.main() @` $\mathbf{n}$ `:` $\mathcal{G}_A^{\mathtt{fin}}$ `= x]` for all $\mathbf{n}$ with $\mathcal{G}_A^{\mathbf{n}}$ `= g`. Note that a priori we do not know that there is such a probability, because probability could depend on $\mathbf{n}$. To show that `(Q g x)` is a well-defined we need to prove that the value `(Q g x)` satisfies the predicate `(P g x)`. Because in the lemma, `(D g)` is only used for `g` of the form $\mathcal{G}_A^{\mathbf{m}}$, we specifically need to show the following claim:

```
have Q_well_def : P $\mathcal{G}_A^{\mathbf{m}}$ x (Q $\mathcal{G}_A^{\mathbf{m}}$ x).
```

(Here we use the EasyCrypt tactic `have name : fact` which allows to locally prove the `fact` and call it `name`.)

If we can show that there exists a probability `q`, so that `P` $\mathcal{G}_A^{\mathbf{m}}$ `x q` then the proof `Q_well_def` amounts to a simple application of the `choicebP` property. The obvious candidate for this probability `q` is the `Pr`-expression:

```
Pr[A.main() @ m : $\mathcal{G}_A^{\mathtt{fin}}$ = x].
```

To show that this candidate satisfies `(P` $\mathcal{G}_A^{\mathbf{m}}$ `x)`, we must prove the following (by definition of `P`):

```
have good_q : ∀ n, $\mathcal{G}_A^{\mathbf{n}}$ = $\mathcal{G}_A^{\mathbf{m}}$ ⇒
    Pr[A.main() @ m : $\mathcal{G}_A^{\mathtt{fin}}$ = x]
  = Pr[A.main() @ n : $\mathcal{G}_A^{\mathtt{fin}}$ = x].
```

The `good_q` is intuitively simple and we prove it using the pRHL which is available in EasyCrypt.

Now, as we know that `(Q g)` assigns adequate probabilities to elements of type $\mathcal{G}_A$, we use the standard

EasyCrypt constructor `mk` which turns any function of type `'a → real` into distribution of type `'a 𝒟`.

```
pose D g := mk (Q g).
```

The above defines a parameterized distribution `D` typed as $\mathcal{G}_A \to \mathcal{G}_A\ \mathcal{D}$. We skip the technical details of a proof which shows that `D` is a well-formed probability distribution. We only show the final derivation which proves that `D` is the denotation of `A.main`. To achieve this we show the point-wise equality of distribution `D` and `Pr`-expression. Let `x` be an element of type $\mathcal{G}_A$:

```
have pointwise :
   μ1 (D G^m_A) x = Pr[A.main() @ m : G^fin_A = x].
```

The proof is as follows:

```
 μ1 (D G^m_A) x
 = μ1 (mk (Q G^m_A)) x
 = Q G^m_A x
 = choiceb (P G^m_A x)
 = Pr[A.main() @ m : G^fin_A = x].
```

The first equality is by definition of `D`, in the second equality μ_1 cancels application of `mk` since `D` is a well-defined distribution (see `muK` from EasyCrypt standard library), the third equality is by definition of `Q`, the fourth equality is an application of the proved property `Q_well_def`.

At the first glance, it seems that this implies that $(\mathtt{D}\ \mathcal{G}^{\mathbf{m}}_A)$ indeed describes the probability distribution corresponding to `A.main`. That is, we want:

```
have onsubs :
   μ (D G^m_A) M = Pr[A.main() @ m : M G^fin_A].
```

meaning that the probability that a value sampled from $(\mathtt{D}\ \mathcal{G}^{\mathbf{m}}_A)$ satisfies `M` equals the probability that the final state of `A.main` satisfies `M`. Unfortunately, this is not immediate. For example, hypothetically, the function $\mathtt{M} \mapsto \mathtt{Pr[A.main()\ @\ \mathbf{m}\ :\ M}\ \mathcal{G}^{\mathtt{fin}}_A]$ might not be a discrete probability measure and thus it might not be determined by its values on singleton sets. To show `onsubs`, we need to use one more trick: We define an auxiliary module and procedure `P.sampleFrom` such that `P.sampleFrom(d)` simply returns some $\mathtt{x} \xleftarrow{\$} \mathtt{d}$. Then $(\mu\ (\mathtt{D}\ \mathcal{G}^{\mathbf{m}}_A)\ \mathtt{M})$ is $\mathtt{Pr[x \leftarrow P.sampleFrom(D}\ \mathcal{G}^{\mathbf{m}}_A)\ \mathtt{@}\ \mathbf{m}\ :\ \mathtt{M\ x]}$. So, `onsubs` becomes:

```
have aux1 : Pr[x ← P.sampleFrom(D G^m_A) @ m : M x]
  = Pr[A.main() @ m : M G^fin_A].
```

For goals of this shape, we use a combination of the `byequiv` and `bypr` tactics from EasyCrypt; `byequiv` changes this goal into a pRHL judgment relating the programs `P.sampleFrom` and `A.main`. And `bypr` converts such a pRHL judgment back to an equality of probabilities. It seems that we are back at `onsubs` now. However, the final equality is actually:

```
have aux2 : ∀ x,
  Pr[r ← P.sampleFrom(D G^m_A) @ 1 : r = x]
  = Pr[A.main() @ 2 : G^fin_A = x].
```

(for memories **1** and **2** that are equal to **m** in global variables $\mathcal{G}^{\mathbf{m}}_A$.) But this follow from `pointwise` proven above. □

Note that in our proof we rely on the fact that the tactics `byequiv` and `bypr` in combination imply that `Pr[...: M x]` can be related to $\mathtt{y} \mapsto$ `Pr[...: x = y]`, even for infinite `M`. This is fortunate because these facts are not accessible in EasyCrypt as explicit lemmas. (For all we know, `Pr[...]` might have defined a content (in the measure-theoretic sense) instead of a measure without contradicting any of the tactics and theorems derivable in EasyCrypt. Our proof shows that this is not the case.)

3.2. Probabilistic Toolkit

In this section, we present well-known results from probability theory which we formalized and used extensively in our EasyCrypt development. In particular, recall that in Thm. 1.1 we used averaging and Jensen's inequality in the derivation of the key lemma needed for proving the security of a coin-toss protocol (see Sec. 5). In turn, proofs of Jensen's inequality and averaging depend on finite probabilistic approximation. (To the best of our knowledge, these results were not previously formalized in EasyCrypt.)

Finite Pr-approximation. We prove that the support of a distribution can be finitely approximated with arbitrary precision. We formally prove finite approximation for distributions and then use the probabilistic reflection to extend this result to programs.

Let `d` be a distribution of type `'a 𝒟`. Then there exists a sequence of lists `(L n)` so that the probability that an element sampled from `d` is not in the list `(L n)` converges to 0 (for $n \to \infty$). (This holds for discrete distributions only.)

```
lemma fin_pr_approx_distr_conv ['a] :
  ∀ (d : 'a 𝒟), ∃ (L : int → 'a ℒ),
   convergeto (λ n. μ d (λ x. x ∉ L n)) 0.
```

Having the finite probabilistic approximation for distributions allows us to use the probabilistic reflection mechanism to extend the finite probabilistic approximation to programs. More specifically, let `A.main` be a procedure which takes an argument of type `at` and produces the result of type `rt`. In this case, there exists a sequence of lists `(L n)`, so that the result and the final state produced by `A.main(a)` are not in `(L n)` with probability converging to 0.

```
lemma fin_pr_approx_prog_conv : ∀ m a,
  ∃ (L : int → (rt × G_A) ℒ),
  convergeto
    (λ n. Pr[r ← A.main(a) @ m : (r, G_A) ∉ L n])
    0.
```

Averaging. Averaging allows us to express the probability corresponding to a program $\mathtt{x} \xleftarrow{\$} \mathtt{d};\ \mathtt{A.main(x)}$ in terms of probabilities corresponding to a program `A.main(x)` and probability assigned to `x` by distribution `d`. In this sense, the averaging technique can be seen as a generalized version of case-analysis.

In our EasyCrypt formalization we state and prove a general version of averaging for an arbitrary distribution `d : rt 𝒟` which might have an infinite support:

```
lemma averaging : ∀ m M i d,
 Pr[x <$ d; r ← A.main(x,i) @ m : M r]
 = sum (λ x. μ1 d x · Pr[r ← A.main(x,i) @ m: M r]).
```

Here, the value `sum f` denotes $\Sigma_x f(x)$. (Note that the sum in EasyCrypt does not have to range over a finite set.)

Jensen's inequality. Jensen's inequality is another well-known result which is widely used in cryptography. In general, it relates the value of a convex function of an integral (or sum) to the integral (or sum) of the convex function. In the context of probability theory, it is generally stated in the following form: if `X` is a distribution, `g` maps elements of `X` to reals, and `f` is a convex function, then `f (E X g)` $\leq$ `E X (f ∘ g)`. Here, `E X h` is an expected value and $\circ$ denotes function composition.

We prove a slightly restricted version of Jensen's inequality. In particular, we assume that on the support of `X` the function `g` takes values in an interval between some parameter-values `a` and `b` and that the `f x` takes values in an interval between parameter-values `c` and `d` if `a` $\leq$ `x` $\leq$ `b`. Also, the standard assumptions are that `f` is convex, that the distribution `X` is lossless (i.e., `μ X (λ x. true) = 1`), and that the expectations `E X g` and `E X (f ∘ g)` exist.

```
lemma Jensen_inf ['a] :
  ∀ (X : 'a 𝒟) g f (a b c d : real),
  is_lossless X
  ⇒ hasE X g
  ⇒ hasE X (f ∘ g)
  ⇒ (∀ (a b : real), (convex f a b))
  ⇒ (∀ x, a ≤ x ≤ b ⇒ c ≤ f x ≤ d)
  ⇒ (∀ x, x ∈ X ⇒ a ≤ g x ≤ b)
  ⇒ f (E X g) ≤ E X (f ∘ g).
```

(The EasyCrypt standard library derives Jensen's inequality for distributions with *finite* support only.)

3.3. Reflection of Composition

In this section we address the probabilistic reflection of the sequential composition of programs. For example, let us analyze the following program: `r1 ← A.ex1(); A.ex2(r1)`. We can use the `reflection_simple` lemma from Sec. 3.1 to get access to a distribution `D12` such that:

```
∀ m M, μ (D12 𝒢_A^m) M
  = Pr[r1 ← A.ex1(); A.ex2(r1) @ m : M 𝒢_A^fin].
```

The distribution `D12` corresponds to a composite program as a whole. However, being able to reflect the distribution corresponding to a composite program is not enough to enable reasoning about composite programs based on the properties of its components; we do not know how `D12` is related to `A.ex1` and `A.ex2` separately.

In the following, we prove a lemma for reflection of composition which allows us to show that there exist distributions `D1` and `D2` which are the probabilistic reflection of procedures `A.ex1` and `A.ex2`, and that the composition of `D1` and `D2` is `D12`. So, the main goal is to prove lemmas that allow us to derive the distribution of a more complex program from the distributions which correspond to its components.

In EasyCrypt, the composition of distributions is implemented as an operator `dlet` which has the following type: `'a 𝒟 → ('a → 'b 𝒟) → 'b 𝒟`. Intuitively, the distribution `dlet d1 d2` could be described as the following imperative program: $x_1 \overset{\$}{\leftarrow} d_1$`;` $x_2 \overset{\$}{\leftarrow} d_2\ x_1$`; return x2`.

We can formally state the theorem of reflection of composition as follows:

```
lemma refl_comp_simple :
 ∃ (D1 : 𝒢_A → (rt1 × 𝒢_A) 𝒟)
   (D2 : 𝒢_A → rt1 → 𝒢_A 𝒟),
 (∀ m M, μ (D1 𝒢_A^m) M
    = Pr[r1 ← A.ex1() @ m : M r1]) ∧
 (∀ m M r1, μ (D2 (r1, 𝒢_A^m)) M
    = Pr[r2 ← A.ex2(r1) @ m : M r2]) ∧
  ∀ m M, μ (dlet (D1 𝒢_A^m) D2) M
    = Pr[r1 ← A.ex1(); A.ex2(r1) @ m : M 𝒢_A^fin].
```

We give only a rough sketch of the proof. First, by using the `reflection` lemma from Sec. 3.1 we get distributions `D1` and `D2` which correspond to procedures `A.ex1` and `A.ex2`, respectively. Next, we use pRHL reasoning to prove that the imperative composition of `D1` and `D2` corresponds to composition of `A.ex1` and `A.ex2`:

```
 Pr[x1 <$ D1; x2 <$ D2 x1 @ m : M x2]
 = Pr[r1 ← A.ex1(); A.ex2(r1) @ m : M 𝒢_A^fin].
```

Finally, we prove that the imperative composition of `D1` and `D2` corresponds to their declarative composition, namely, `dlet D1 D2`:

```
Pr[x1 <$ D1; x2 <$ D2 x1 @ m : M 𝒢_A^fin]
 = μ (dlet (D1 𝒢_A^m) D2) M.
```

This step uses averaging (see Sec. 3.2).

In our EasyCrypt formalization we prove a more powerful lemma `refl_comp` which generalizes `refl_comp_simple` in the following aspects:

- The procedures `A.ex1` and `A.ex2` take all-quantified arguments `i1` of type `at1` and `i2` of type `at2`, respectively. As a result, the distributions `D1` and `D2` also become parameterized by values of types `at1` and `at2`, respectively.
- The distribution `(D2 i g)` is over pairs `(r, 𝒢_A^fin)` of output of `A.ex2` and final memory (not just the final memory).
- In the event part of the probability expression (i.e., `Pr[...: M (r1, r2, 𝒢_A^fin)]`) we allow the predicate `M` to depend on the `r1` (output of `A.ex1`), `r2` (output of `A.ex2`), and the final memory $\mathcal{G}_A^{\mathtt{fin}}$ (not just the final memory).

In EasyCrypt, we prove the following general version of reflection of composition:

```
lemma refl_comp :
 ∃ (D1 : at1 → 𝒢_A → (rt1 × 𝒢_A) 𝒟)
   (D2 : at2 → rt1 × 𝒢_A → (rt2 × 𝒢_A) 𝒟),
 (∀ m M i1, μ (D1 i1 𝒢_A^m) M
   = Pr[r1 ← A.ex1(i1) @ m : M (r1, 𝒢_A^fin)]) ∧
 (∀ m M r1 i2, μ (D2 i2 (r1, 𝒢_A^m)) (M r1)
   = Pr[r2 ← A.ex2(i2, r1) @ m : M r1 (r2, 𝒢_A^fin)]) ∧
  ∀ m M i1 i2, μ (dlet (D1 i1 𝒢_A^m)
       (λ r. dmap (D2 i2 r) (λ x. (r.1, x)))) M
   = Pr[r1 ← A.ex1(i2); r2 ← A.ex2(i2, r1)
       @ m : M (r1, r2, 𝒢_A^fin)].
```

Here `dmap d f` denotes the distribution of `f x` for $x \overset{\$}{\leftarrow} d$.

4. Rewinding

In Sec. 1, we briefly explained that rewinding is a commonly used technique which allows one module (i.e., program) to save a state of another module and also restore that state at some later time. More precisely, we say that a module `A` is rewindable iff:

1) There exists an injective mapping `f` from $\mathcal{G}_A$ to some parameter type `sbits`[2].
2) The module `A` must have a terminating procedure `getState`, so that whenever `A.getState` is called from the state `g` : $\mathcal{G}_A$, the result of the call must be equal to `(f g)` and the state of `A` must not change.
3) The module `A` must have a terminating procedure `setState`, so that whenever it is given an argument `x : sbits` so that `x = (f g)` for some `g` : $\mathcal{G}_A$ then `A` must be set into a state `g`.

In EasyCrypt, we start formalizing this definition by defining a module type `Rew` for rewindable programs:

```
module type Rew = {
  proc getState() : sbits
  proc * setState(s : sbits) : unit
}.
```

(Here, the symbol `*` next to the procedure `setState` tells EasyCrypt that after the execution of that procedure all global variables of a module must be (re)initialized.)

We formalize the rewinding properties of `getState` and `setState` procedures as a predicate `RewProp` on modules typeable as `Rew`. Unfortunately, in EasyCrypt we cannot define an operator like `RewProp` because its definition depends on a module which is not allowed. As a result, in the actual EasyCrypt code the workaround is to copy-and-paste the verbose definition of `RewProp(A)`. This reduces readability, but is conceptually the same.

```
op RewProp(A : Rew) : bool =
  ∃ (f : 𝒢_A → sbits), injective f ∧
   (∀ m, Pr[r ← A.getState() @ m
           : 𝒢_A^fin = 𝒢_A^m ∧ r = f 𝒢_A^m] = 1) ∧
   (∀ m (g : 𝒢_A), Pr[A.setState(f g) @ m
           : 𝒢_A^fin = g] = 1) ∧
  islossless A.setState.
```

(In EasyCrypt, `islossless X.p` expresses that the procedure `p` of module `X` must terminate on all inputs.)

Then, whenever we do a proof using rewinding, we will need to explicitly assume that our adversary `A` satisfies `RewProp(A)`, or, equivalently, quantify only over adversaries of module type `Rew` satisfying `RewProp(A)`.

The first litmus test of our definition of rewindability is to show that modules without global variables are (trivially) rewindable. For a module without global variables, $\mathcal{G}_A$ will be a singleton type (i.e., ∀(x y : $\mathcal{G}_A$), x = y). Then we can generically state that `A` will be rewindable, as long as we implement `getState` and `setState` as terminating procedures (it does not matter what they do):

```
lemma no_globs_rew : ∀ (A <: Rew),
 (∀ (x y : 𝒢_A), x = y)
  ⇒ islossless A.getState ∧ islossless A.setState
  ⇒ RewProp(A).
```

2. Intuitively, `sbits` is the type of bitstrings. To keep our development more general, we do not require this, but only assume the existence of embeddings `nat_sbits: nat → sbits` (to ensure that `sbits` is infinite) and `pair_sbits: sbits × sbits → sbits`. The EasyCrypt theory cloning mechanism makes it possible to later replace this `sbits` type by a concrete type such as lists of bits.

On the necessity of the `RewProp` axiom. The reader may wonder whether adding the explicit assumption in a security proof that the adversary `A` satisfies `RewProp(A)` does not weaken the security proof. After all, it means that security only holds with respect to such adversaries, but not with respect to adversaries that do not satisfy `RewProp(A)`. We argue that `RewProp(A)` is not a true restriction of the adversary, merely a requirement that the adversary has a certain interface with certain properties. The only actual restriction about the inner workings of the adversary that `RewProp(A)` makes is that the adversary's state can be encoded as a sequence of bits (`sbits`). Usually, in cryptography, we make even stronger assumptions about the adversary, namely that its state *is* a sequence of bits (or a Turing machine tape). In contrast, here we only assume that its state *can be encoded* as a sequence of bits.

We stress that we only need to make this assumption for abstract (i.e., all-quantified) adversaries. For adversaries that we explicitly construct as part of a reduction, we can actually prove `RewProp`, see the next section.

4.1. Transformations

Cryptographic proofs are commonly based on transformations of adversaries (or reduction of adversaries). In EasyCrypt, a transformation is a module which receives other modules as parameters, defines its own global variables, and has procedures which can call procedures of its parameter-modules. Typically, one of the parameter-modules will be the original adversary.

In this section, we show how to prove rewindability of a module which is parameterized by rewindable modules and which has at most countable state. We illustrate this by implementing a module `T` which is parameterized by rewindable modules `A` and `B` and has a global variable `x` of a parameter type `ct`. As a result, the global state of module `T(A,B)` consist of variable `T.x` and all global variables of modules `A` and `B`. (i.e., $\mathcal{G}_{T(A,B)}$ = ($\mathcal{G}_A$ × $\mathcal{G}_B$ × `ct`)). Since, by the definition of rewindability, we need to embed elements of type $\mathcal{G}_{T(A,B)}$ into `sbits` then we parameterize our development by an injection from `ct` to `sbits`:

```
op ct_sbits : ct → sbits.
axiom bcu   : injective ct_sbits.

module T(A : Rew, B : Rew) : Rew = {

  var x : ct

  // add your own procedures here

  proc getState(): sbits = {
    var stateA, stateB, xsbits : sbits;
    stateA ← A.getState();
    stateB ← B.getState();
    xsbits ← ct_sbits x;
    return pair_sbits
      (xsbits, pair_sbits(stateA, stateB));
  }

  proc setState(state: sbits): unit = {
    var stateA, stateB, xsbits : sbits;
    (xsbits, s) ← unpair_sbits state;
    (stateA, stateB) = unpair_sbits s;

     A.setState(stateA);
```

```
      B.setState(stateB);
      x ← unct_sbits xsbits;
   }
}.
```

The procedure getState stores the global states of A and B in the local variables stateA and stateB, respectively. Then the global variable T.x is converted into sbits and saved in variable xsbits. The resulting state is an embedding of a nested tuple (xsbits,(stateA,stateB)) into sbits. (Recall that pair_sbits is an embedding sbits × sbits → sbits.)

The procedure setState receives an sbits argument which is then "untupled" into sbits variables xsbits, stateA, and stateB. The state of A is set by passing argument stateA to its implementation of setState procedure (similarly for B, *mutatis mutandis*). The variable T.x is set to the preimage of xsbits. Finally, we use pHL to prove that T(A,B) is also rewindable:

```
lemma trans_rew : ∀ (A :> Rew) (B :> Rew{A}),
   RewProp(A) ∧ RewProp(B) ⇒ RewProp(T(A,B)).
```

Note, that in the statement of the lemma we additionally require a state of B to be disjoint from a state of A (i.e., B :> Rew{A}). This is required because in case of possibly overlapping states not all values of type $\mathcal{G}_{T(A,B)}$ are valid states. For example, if $\mathcal{G}_A^{\mathbf{m}}$ and $\mathcal{G}_B^{\mathbf{n}}$ have overlapping variables with different values then the value $(\mathcal{G}_A^{\mathbf{m}}, \mathcal{G}_A^{\mathbf{n}}, \mathtt{x})$ is typeable as $\mathcal{G}_{T(A,B)}$ (for any x of type ct), but does not represent a possible state of T(A,B). Unfortunately, in EasyCrypt, it is not possible to express "consistency" of possibly overlapping states of abstract modules.

4.2. Multiplication Rule and Commutativity

The multiplication rule from probability theory states that the probability of independent events occurring simultaneously is found by multiplying the probabilities of each event.

In terms of probabilistic programs it is natural to say that an execution of a procedure P.run is independent of an execution of Q.run if after termination of P.run the state of Q is not affected. In EasyCrypt, it is easy to prove the *multiplication rule* for modules with disjoint states:

```
lemma rew_mult_simple : ∀ (P :> Runnable)
     (Q :> Runnable{P}) m M1 M2 i1 i2,
 Pr[r1 ← P.run(i1); r2 ← Q.run(i2) @ m
   : M1 r1 ∧ M2 r2]
  = Pr[ r1 ← P.run(i1) @ m : M1 r1]
  · Pr[ r2 ← Q.run(i2) @ m : M2 r2].
```

However, if we need independent runs of the procedure(s) of the same module then we need rewinding. Recall, that the main goal of rewindability is to be able to restore the state of a module after running one of its procedures. Let A be a rewindable module (i.e., A satisfies RewProp(A)) with procedures $\mathtt{ex}_1$ and $\mathtt{ex}_2$ which take all-quantified arguments $\mathtt{i}_1 : \mathtt{at}_1$ and $\mathtt{i}_2 : \mathtt{at}_2$ and compute results $\mathtt{r}_1 : \mathtt{rt}_1$ and $\mathtt{r}_2 : \mathtt{rt}_2$, respectively. Let us analyze the following program.

(1) Save the initial state of A by calling s ← A.getState().
(2) Run a procedure $\mathtt{r}_1$ ← $\mathtt{A.ex}_1(\mathtt{i}_1)$.
(3) Restore the initial by calling A.setState(s).
(4) Run a procedure $\mathtt{r}_2$ ← $\mathtt{A.ex}_2(\mathtt{i}_2)$.

First, we analyze the steps (1)–(3) as a standalone program. In particular we must show that the getState and the setState calls do not affect the result computed by $\mathtt{A.ex}_1$ procedure and also show that the final state of A equals to its initial state.

```
lemma rew_clean : ∀ m M1 i1,
  Pr[s ← A.getState(); r1 ← A.ex1(i1);
     A.setState(s) @ m : M1 r1 ∧ G^m_A = G^fin_A]
  = Pr[r1 ← A.ex1(i1) @ m : M1 r1].
```

(The proof requires only basic pHL tactics and rewindability axioms.) This result allows us to derive the *multiplication rule* which states that the probability of a joint event $\mathtt{M}_1\ \mathtt{r}_1 \wedge \mathtt{M}_2\ \mathtt{r}_2$ for the program (1)–(4) on memory **m** equals to the product of probabilities of events $\mathtt{M}_1\ \mathtt{r}_1$ and $\mathtt{M}_2\ \mathtt{r}_2$ occurring after independent runs of $\mathtt{A.ex}_1$ and $\mathtt{A.ex}_2$ on **m**, respectively. In EasyCrypt this is stated as follows:

```
lemma rew_mult_law : ∀ m M1 M2 i1 i2,
  Pr[s ← A.getState(); r1 ← A.ex1(i1);
     A.setState(s); r2 ← A.ex2(i2) @ m
    : M1 r1 ∧ M2 r2]
  = Pr[r1 ← A.ex1(i1) @ m : M1 r1]
   · Pr[r2 ← A.ex2(i2) @ m : M2 r2].
```

In its essence, rew_mult_law is derived by a single call to the built-in seq tactic.

Commutativity. In its turn, the multiplication rule opens for us an easy route to proving commutativity for rewindable modules. Consider a program consisting of steps (1)–(4)–(3)–(2) (i.e., $\mathtt{A.ex}_1$ and $\mathtt{A.ex}_2$ calls are swapped). We can prove that it computes the same distribution of pairs $(\mathtt{r}_1, \mathtt{r}_2)$ as the program (1)–(4).

```
lemma rew_comm_law_simple : ∀ m M i1 i2,
    Pr[s ← A.getState(); r1 ← A.ex1(i1);
       A.setState(s); r2 ← A.ex2(i2) @ m
     : M (r1, r2)]
    = Pr[s ← A.getState(); r2 ← A.ex2(i2);
         A.setState(s); r1 ← A.ex1(i1) @ m
        : M (r1, r2)].
```

By using the combination of byequiv and bypr tactics we reduce the above lemma to a point-wise equality of programs:

```
have aux1 : ∀ x1 x2,
    Pr[s ← A.getState(); r1 ← A.ex1(i1);
       A.setState(s); r2 ← A.ex2(i2) @ m
      : r1 = x1 ∧ r2 = x2]
    = Pr[s ← A.getState(); r2 ← A.ex2(i2);
         A.setState(s); r1 ← A.ex1(i1) @ m
        : r1 = x1 ∧ r2 = x2].
```

Then:

```
have aux2 : ∀ x1 x2,
    Pr[s ← A.getState(); r1 ← A.ex1(i1);
       A.setState(s); r2 ← A.ex2(i2) @ m
      : r1 = x1 ∧ r2 = x2]
  = Pr[r1 ← A.ex1(i1) @ m : r1 = x1] // rew_mult
  · Pr[r2 ← A.ex2(i2) @ m : r2 = x2]
  = Pr[r2 ← A.ex2(i2) @ m : r2 = x2] // comm of ·
  · Pr[r1 ← A.ex1(i1) @ m : r1 = x1]
  = Pr[s ← A.getState(); r2 ← A.ex2(i2);
       A.setState(s); r1 ← A.ex1(i1) @ m
     : r1 = x1 ∧ r2 = x2].  // rew_mult
```

(In the second invocation of rew_mult, the procedure names $\mathtt{ex}_1$ and $\mathtt{ex}_2$ are exchanged. Since lemmas in EasyCrypt are not parametric in the procedure names, we achieve this by using a wrapper module.)

In the actual EasyCrypt formalization, we prove a slightly more general version of commutativity for rewindable modules. In particular, we allow the program to start with a call to B.init (which might not be disjoint from A). As a result of this change, the proof starts by reflecting the composition of B.init with the rest of the program and then using the lemma rew_comm_law_simple.

```
lemma rew_comm_law : ∀ m M i0,
  Pr[r0 ← B.init(i0); s ← A.getState();
     r1 ← A.ex1(r0); A.setState(s);
     r2 ← A.ex2(r0) @ m : M (r0, r1, r2)]
= Pr[r0 ← B.init(i0); s ← A.getState;
     r2 ← A.ex2(r0); A.setState(s);
     r1 ← A.ex1(r0) @ m : M (r0, r1, r2)].
```

Note that the reflection of composition relies on "averaging technique" which relies on finite probabilistic approximation (see Sec. 3.2).

4.3. Rewinding with Initialization

In Thm. 1.1 we sketched a derivation of the equation which is needed to prove sum-binding property for commitments (see Sec. 5). More specifically, we analyzed a program which starts with an explicit state initializer, saves the resulting state of module A, runs a procedure A.run for the first time, restores the saved state, and then runs the A.run procedure for the second time. We proved that the probability of a success (according to some predicate) in two sequential runs of A.run is lower-bounded by a square of probability of a success in a "initialize-then-run" case (i.e., initialize the state and execute the A.run procedure once).

In EasyCrypt, we derive a similar equation, but for a more general case:

- The initialization is done with a procedure B.init, where B is a module with a state which can possibly intersect with the state of module A.
- The initialization produces a result $\mathtt{r}_0$ of a parameter type which is then supplied to A.run.
- The procedures B.init receives all-quantified argument i of a parameter type.
- The procedure A.run returns a result of a parameter type rt. The success of a run is defined by a parameter predicate $\mathtt{M}\ (\mathtt{r}_0,\ \mathtt{r}_i)$, where $\mathtt{r}_0$ and $\mathtt{r}_i$ are the values returned by B.init and A.run procedures, respectively.

The EasyCrypt statement of the lemma is as follows:

```
lemma rew_with_init : ∀ m M i,
 Pr[r0 ← B.init(i); s ← A.getState();
    r1 ← A.run(r0); A.setState(s);
    r2 ← A.run(r0) @ m : M (r0, r1) ∧ M (r0, r2)]
≥ Pr[r0 ← B.init(i); r ← A.run(r0) @ m
    : M (r0, r)]^2.
```

We skip the proof as it roughly follows the steps sketched in Thm. 1.1.

5. Case Study: Commitments and Coin-Toss

As a case study for our techniques we prove the security of a coin-toss protocol based on bit-commitment. Historically, Blum described the problem of coin-toss protocol with the following example: *Alice and Bob are recently divorced, living in two separate cities, and want to decide who gets to keep the car. To decide, Alice wants to flip a coin over the telephone. However, Bob is concerned that if he were to tell Alice heads, she would flip the coin and automatically tell him that he lost.* Thus, the problem with Alice and Bob is that they do not trust each other; the only resource they have is the telephone communication channel, and there is not a third party available to read the coin [13].

In the following, we describe the coin-toss protocol based on a bit-commitment scheme which is similar to the original Blum's solution to the coin-toss problem:

1) Alice chooses a random bit $\mathtt{r}_1$ and then generates a commitment c containing that bit (let d be the respective opening).
2) Alice sends the commitment c to Bob.
3) Bob chooses a random bit $\mathtt{r}_2$ and sends it to Alice.
4) Alice opens her commitment by sending the bit $\mathtt{r}_1$ and the opening d to Bob.
5) Bob verifies that d is a valid opening of $\mathtt{r}_1$ for c. Otherwise Bob aborts.
6) Alice and Bob compute the final bit as $\mathtt{r}_1 \oplus \mathtt{r}_2$ (xor).

The coin-toss protocol must ensure the following property: if at least one of the parties correctly generates a random bit, then the final bit will be (nearly) random.

Security of the coin-toss is almost immediate if the commitment scheme satisfies a property called "sum-binding" in [14]. This property says that the probability of Alice opening the commitment to false and the probability of Alice opening it to true add to at most 1 (plus a negligible error). This property in turn is implied by the usual "computationally binding" property which says that Alice cannot open to both false and true *simultaneously* (except with negligible probability). Showing that "computationally binding" implies "sum-binding", however, requires rewinding. Therefore that proof is a prime candidate for our case-study. (In the post-quantum setting, for example, computationally binding does *not* imply sum-binding [15]. This illustrates that this seemingly trivial implication is not as easy as it might seem, and that we indeed need rewinding here.)

5.1. Commitments

In the standard library of EasyCrypt, the module type CommitmentScheme requires a commitment scheme S to implement the following procedures:

1) p ← S.gen() generates the public-key of a commitment scheme (also known as the public parameters).
2) (c, d) ← S.commit(p, m) produces a commitment c and an opening d for a message m and a public-key p.

3) b ← S.verify(p, m, c, d) returns b = true iff d is a valid opening for message m, commitment c, and public-key p.

For our development, we additionally require the existence of a verification function Ver (an "operator" in EasyCrypt-parlance) which must agree with the procedure S.verify on all arguments:

```
op Ver : pubkey × message × commitment × opening
  → bool.

axiom verify_det : ∀ m a,
  Pr[r ← S.verify(a) @ m : r = Ver a] = 1.
```

This means that verification is side-effect free (and deterministic). Otherwise, two runs of the verification algorithm could interfere with each other (and with calls to S.commit) and give different results.

In cryptography, a commitment scheme is called *computationally binding* iff the probability that adversary A can produce a commitment with openings of two different messages is negligible. The EasyCrypt standard library defines a module type Binder with a single procedure bind; we can then define the probability of success of adversary A : Binder in the "binding-game":

```
op binding_pr(A, m) = Pr[p ← S.gen();
  (c, m1, d1, m2, d2) ← A.bind(p);
   v1 ← S.verify(p,m1,c,d1);
   v2 ← S.verify(p,m2,c,d2) @ m
  : v1 ∧ v2 ∧ m1 ≠ m2].
```

(Here, binding_pr is only a shortcut notation used in this text.) Hence, scheme is binding if binding_pr(A,**m**) is negligible for all A and **m**.

Sum-Binding. Next, we define the "sum-binding" property of commitments. Let A be an adversary and p_b be a probability that A can open the commitment to contain b given input b = false,true. The commitment scheme is *sum-binding* iff for all such adversaries the $p_f + p_t \leq 1 + \epsilon$, where ϵ is negligible. We define a module type SumBinder with procedures commit and open. Then we define the probability of success of adversary A : SumBinder in the "sum-binding-game":

```
op sum_binding_pr(A, m)  =
 Pr[p ← S.gen(); c ← A.commit(p);
      d ← A.open(0); v ← S.verify(p,0,c,d)
   @ m : v]
 + Pr[p ← S.gen(); c ← A.commit(p);
      d ← A.open(1); v ← S.verify(p,1,c,d)
   @ m : v].
```

(Again, sum_binding_pr is only a shortcut notation.) Hence, scheme is sum-binding if and only if for all A and **m**, there exists a negligible ϵ, so that sum_binding_pr(A, **m**) $\leq 1 + \epsilon$. Before addressing the sum-binding property for commitments, we prove a more generic sum-binding inequality which shows that the sum of probabilities of success of independent runs of arbitrary procedures A.ex$_1$ and A.ex$_2$ is related to the probability of joint success in the same run.[3] More specifically, assume that module A is rewindable and B.init is some initialization procedure. We let p_1 be the probability that after initialization the procedure A.ex$_1$ succeeds according to some predicate M (similarly for p_2 and A.ex$_2$, *mutatis mutandis*). In this case, we can prove that the sum of probabilities $p_1 + p_2$ is upper-bounded by a sum $1 + 2 \cdot q$, where q is the probability that A.ex$_1$ and A.ex$_2$ both succeed in the same run (i.e., both starting from the same initial state produced by B.init). In EasyCrypt, we state this equation as follows:

```
axiom rewindable_A : RewProp(A).

lemma sum_binding_generic : ∀ m M i,
   Pr[r0 ← B.init(i); r ← A.ex1(r0) @ m : M r]
 + Pr[r0 ← B.init(i); r ← A.ex2(r0) @ m : M r]
   ≤ 1 + 2 · Pr[r0 ← B.init(i); s ← A.getState();
               r1 ← A.ex1(r0); A.setState(s);
               r2 ← A.ex2(r0) @ m : M r1 ∧ M r2].
```

Proof: Let us define the following shortcut-notation:

```
Pj  = Pr[r0 ← B.init(i); r ← A.exj(r0) @ m : M r].
Pjk = Pr[r0 ← B.init(i); s ← A.getState();
         r1 ← A.exj(r0); A.setState(s);
         r2 ← A.exk(r0) @ m : M r1 ∧ M r2].
P$  = Pr[r0 ← B.init(i); j ←$ {1,2};
         r ← A.exj(r0) @ m : M r].
P$$ = Pr[r0 ← B.init(i);
         s ← A.getState(); j ←$ {1,2};
         r1 ← A.exj(r0); A.setState(s); k ←$ {1,2};
         r2 ← A.exk(r0) @ m : M r1 ∧ M r2].
```

Here, P_j denotes a probability of success of a run of a procedure A.ex$_j$ (in EasyCrypt, we implement A.ex$_j$ using the if-then-else construct). P_{jk} denotes a probability of a joint success of a run of procedures A.ex$_j$(r_0) and A.ex$_k$(r_0) from the same initial state. $P_\$$ denotes a success of a run of a procedure A.ex$_j$ where j is sampled uniformly from {1,2}. Finally, $P_{\$\$}$ denotes a probability of a joint success of a run of procedures A.ex$_j$ and A.ex$_k$ where both j and k are uniformly sampled from {1,2}.

Using our notation, the statement of the lemma sum_binding_generic can be therefore expressed as:

```
have goal : P1 + P2 ≤ 1 + 2 · P12.
```

Before continuing with the proof we list some basic facts about these definitions:

```
have f1 : P$  = 1/2 · (P1 + P2).
     f2 : P$$ = 1/4 · (P11 + P12 + P21 + P22).
     f3 : ∀ x y, Px  ≥ Pxy.
     f4 : P$$ ≥ P$^2.
```

The facts f_1 and f_2 are by case analysis. The fact f_3 is by event inclusion. The fact f_4 is by rewinding with initialization equation rew_with_init derived in Sec. 4.3.

To prove the goal we first derive an equation which connects P_{12} and P_{21} to P_1 and P_2:

```
have aux : P12 + P21 ≥ P1 + P2 - 1.
```

To prove aux we argue as follows:

```
P12 + P21
= 4 · (1/4 · (P12 + P21 + P11 + P22)
    - 1/4 · (P11 + P22))          // math
= 4 · (P$$ - 1/4 · (P11 + P22))   // f2
```

3. This generic lemma may also be useful when analyzing certain extractors for proof of knowledge protocols with two challenges, e.g., the zero-knowledge protocols for Hamiltonian cycles [16] and for graph isomorphism [17].

```
≥ 4 · (P$$ - 1/4 · (P1 + P2))      // f3
= 4 · (P$$ - 1/2 · P$)             // f1
≥ 4 · (P$^2 - 1/2 · P$)            // f4
≥ 2 · P$ - 1                       // math
= P1 + P2 - 1.                     // f1
```

Finally, the `goal` is concluded by using the `aux` inequality and observing that $P_{12} = P_{21}$ (due to commutativity rule `rew_comm_law`, see Sec. 4.2). □

Equipped with the generic sum-binding inequality we can now finish the proof that binding commitment schemes are also sum-binding. We start by implementing a reduction `R(A)` which runs `A.commit` (to produce the commitment), stores the state of `A`, runs `A.open(false)` (to produce the first opening), restores the state of `A`, and runs `A.open(true)` (to produce the second opening). Then we show that the probability that `R(A)` produces two valid openings (i.e., breaks binding) is lower-bounded in terms of the probability that `A` is successful in producing one valid opening.

```
module R(A : SumBinder) : Binder = {
   proc bind(p : pubkey) = {
     var c,s,d1,d2;
     c ← A.commit(p);
     s ← A.getState();
     d1 ← A.open(false);
     A.setState(s);
     d2 ← A.open(true);
     return (c,false,d1,true,d2);
   }
}.
```

Next, we implement wrapper-modules `B` and `A'`, so that `B.init` is a wrapper around "commitment initialization" phase `p ← S.gen(); c ← A.commit(p)`. The procedure `A'.ex1` is defined as `A.open(false)`, and `A'.ex2` as `A.open(true)`. In this case, sum-binding for commitments becomes an immediate consequence of the `sum_binding_generic` inequality and we can conclude:

```
axiom rew_A : RewProp(A).

lemma commitment_sum_binding : ∀ m,
  sum_binding_pr(A, m)
    ≤ 1 + 2 · binding_pr(R(A), m).
```

5.2. Coin-Toss Protocol

Recall, that a coin-toss protocol is considered secure if it is ensured that if at least one of the parties correctly generates a random bit then the final bit will be (nearly) random.

In the first case, we assume that Alice is honest and Bob is cheating. To simplify this case, we additionally assume that the commitment scheme is perfectly hiding. (The case of cheating Bob does not involve rewinding and is therefore not the focus of this paper.) This means that Bob gets no information about r_1 after receiving the commitment `c`. Therefore, if Alice follows the protocol honestly and r_1 is uniformly random and independent of r_2 (due to the perfect hiding) then the resulting bit $(r_1 \oplus r_2)$ is also uniformly random.

In the second case, we are left to show that if Bob honestly follows the protocol, then for any Alice (adversary `A : CoinTossAlice`) the resulting bit is nearly uniform. Below we assume that module type `CoinTossAlice` requires a module to have procedures `commit` and `toss`, where `commit` produces a commitment `c`, and `toss` gets a Bob's bit r_2 as an argument and then computes a bit together with its opening for `c`. We write `coin_toss_pr(A,m,b)` to denote a probability of `A` being able to open the commitment to Boolean `b`.

```
op coin_toss_pr(A,m,b) =
  Pr[p ← S.gen(); r2 <$ {0,1}; c ← A.commit(p);
     (r1,d) ← A.toss(r2) @ m
     : Ver (p,r1,c,d) ∧ r1 ⊕ r2 = b].
```

(Here, `sum_binding_pr` is only a shortcut notation.) We define B_f `(A)` and B_t `(A)` as the transformations of coin-toss adversary into an adversary that breaks binding for the cases `b = false` and `b = true`, respectively:

```
module Bt(A : CoinTossAlice) : SumBinder = {
  proc commit(p : pubkey) = {
    return A.commit(p);
  }
  proc open(x : bit) = {
    var d, r1;
    (r1, d) ← A.toss(not x);
    return d;
  }
}.

module Bf(A : CoinTossAlice) : SumBinder = {
  proc commit(p : pubkey) = {
    return A.commit(p);
  }
  proc open(x : bit) = {
    var d, r1;
    (r1, d) ← A.toss(x);
    return d;
  }
}.
```

B_f `(A)` delegates the commitment generation to `A` and when asked to open a commitment to bit `x` then `x` is submitted to `A.toss` and the resulting opening is returned. B_t `(A)` is different in that the negation of `x` is submitted to `A.toss`. Finally, we can derive that if Bob is honest then for any Alice the resulting bit is nearly uniform.

```
lemma coin_toss_alice : ∀ m b,
 coin_toss_pr(A,m,b)
  ≤ 1/2 + max binding_pr(R(Bt(A)),m)
              binding_pr(R(Bf(A)),m).
```

Proof: We start the proof by analyzing the case when `b = true` (i.e., $r_1 \oplus r_2$ `= true`). We prove that this case is upper-bounded by $1/2 + \epsilon$, where ϵ is the probability of breaking the binding of `S` by `R(`B_t`(A))`.

```
have coin_toss_alice_true :
 coin_toss_pr(A,m,true)
    ≤ 1/2 + binding_pr(R(Bt(A)),m).
```

We prove this case by arguing as follows:

```
Pr[p ← S.gen(); r2 <$ {0,1}; c ← A.commit(p);
  (r1,d) ← A.toss(r2) @ m
  : Ver (p,r1,c,d) ∧ r1 ⊕ r2 = true]
(1)
=  1/2 · (Pr[p ← S.gen(); c ← A.commit(p);
           (r1,d) ← A.toss(false) @ m
           : Ver (p,r1,c,d) ∧ r1 ⊕ false = true]
      +  Pr[p ← S.gen(); c ← A.commit(p);
           (r1,d) ← A.toss(true) @ m
           : Ver (p,r1,c,d) ∧ r1 ⊕ true = true])
(2)
≤  1/2 · (Pr[p ← S.gen(); c ← A.commit(p);
           (r1,d) ← A.toss(false) @ m
```

```
          : Ver (p,1,c,d)]
      +  Pr[p ← S.gen(); c ← A.commit(p);
           (r1,d) ← A.toss(true) @ m
          : Ver (p,0,c,d)])
(3)
 =   1/2 · (Pr[p ← S.gen(); c ← A.commit(p);
            d ← B(A).open(true) @ m
          : Ver (p,false,c,d)]
      +  Pr[p ← S.gen(); c ← A.commit(p);
            d ← B(A).open(false) @ m
          : Ver (p,true,c,d)])
(4)
 ≤   1/2 + binding_pr(R(B_t(A)),m).
```

(Here `{0,1}` denotes a uniform distribution of Booleans.) Step (1) is by case distinction of r_2. Step (2) is by simplification and event-inclusion. Step (3) is by definition of transformation B_t. Step (4) is the application of `commitment_sum_binding` from Sec. 5.1.

In a similar way we handle the case when `b = false` and show:

```
have coin_toss_alice_false :
 coin_toss_pr(A,m,false)
   ≤ 1/2 + binding_pr(R(B_f(A)),m).
```

Finally, `coin_toss_alice` is a trivial consequence of `coin_toss_alice_true` and `coin_toss_alice_false`. □

6. Conclusions

In this paper we focused on probabilistic reflection and rewindability of adversaries. First, we implemented a powerful toolkit for probabilistic reflection which includes finite probabilistic approximation, averaging, and reflection of composition inside EasyCrypt. Second, we described a notion of rewindable adversaries and derived their basic properties: transformations, multiplication rule, commutativity, rewinding with initialization. Third, by combining these results together we were able to derive a generic sum-binding equation for arbitrary rewindable computations. Fourth, we instantiated the sum-binding property for commitments and proved that if a commitment scheme is binding then it is also sum-binding. Finally, we used this result to prove the security of a bit-commitment based coin-toss protocol.

To the best of our knowledge, neither probabilistic reflection, rewindable adversaries, nor security of a coin-toss protocol were not yet addressed in theorem provers.

Acknowledgement

This work was partially supported by the ESF-funded Estonian IT Academy research measure (project 2014-2020.4.05.19-0001) and by the ERC consolidator grant CerQuS (819317), by the Estonian Centre of Excellence in IT (EXCITE) funded by ERDF, by PUT team grant PRG946 from the Estonian Research Council.

Index

References

[1] G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin, "Computer-aided security proofs for the working cryptographer," in *Annual Cryptology Conference*. Springer, 2011, pp. 71–90.

[2] V. Cortier, C. C. Drăgan, F. Dupressoir, B. Schmidt, P.-Y. Strub, and B. Warinschi, "Machine-checked proofs of privacy for electronic voting protocols," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 993–1008.

[3] D. Firsov, H. Lakk, and A. Truu, "Verified multiple-time signature scheme from one-time signatures and timestamping," in *2021 2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. Los Alamitos, CA, USA: IEEE Computer Society, jun 2021, pp. 653–665. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CSF51468.2021.00051

[4] G. Barthe, G. Danezis, B. Grégoire, C. Kunz, and S. Zanella-Béguelin, "Verified computational differential privacy with applications to smart metering," in *2013 IEEE 26th Computer Security Foundations Symposium*, 2013, pp. 287–301.

[5] J. Nussbaumer, "Security analysis for IPsec with EasyCrypt," Master's thesis, University of Bonn, 2019.

[6] D. A. Basin, A. Lochbihler, and S. R. Sefidgar, "CryptHOL: Game-based proofs in higher-order logic," *Journal of Cryptology*, vol. 33, no. 2, pp. 494–566, 2020.

[7] A. Petcher and G. Morrisett, "The foundational cryptography framework," in *International Conference on Principles of Security and Trust*. Springer, 2015, pp. 53–72.

[8] B. Blanchet, "A computationally sound mechanized prover for security protocols," *IEEE Transactions on Dependable and Secure Computing*, vol. 5, no. 4, pp. 193–207, 2008.

[9] M. Berg, "Formal verification of cryptographic security proofs," Ph.D. dissertation, Saarland University, 2013.

[10] G. Barthe, B. Grégoire, and S. Zanella Béguelin, "Formal certification of code-based cryptographic proofs," in *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2009, pp. 90–101.

[11] "Accompanying EasyCrypt development," http://firsov.ee/ec-reflection-rewinding-coin-toss.zip, accessed: 2021-08-21.

[12] G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P.-Y. Strub, "EasyCrypt: A tutorial," in *Foundations of security analysis and design vii*. Springer, 2013, pp. 146–166.

[13] M. Blum, "Coin flipping by telephone: a protocol for solving impossible problems," *ACM SIGACT News*, vol. 15, no. 1, pp. 23–27, 1983.

[14] D. Unruh, "Computationally binding quantum commitments," in *Advances in Cryptology – EUROCRYPT 2016*, M. Fischlin and J.-S. Coron, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 497–527.

[15] A. Ambainis, A. Rosmanis, and D. Unruh, "Quantum attacks on classical proof systems: The hardness of quantum rewinding," in *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, 2014, pp. 474–483.

[16] M. Blum, "How to prove a theorem so no one else can claim it," in *In: Proceedings of the International Congress of Mathematicians*, 1987, pp. 1444–1451.

[17] M. Blum, P. Feldman, and S. Micali, "Non-interactive zero-knowledge and its applications," in *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, ser. STOC '88. New York, NY, USA: Association for Computing Machinery, 1988, p. 103–112. [Online]. Available: https://doi.org/10.1145/62212.62222