Forking Lemma in EasyCrypt

Denis Firsov
Tallinn University of Technology
Input Output
Tallinn, Estonia
denis.firsov@taltech.ee

Jakub Janků
Masaryk University
Brno, Czech Republic
jjanku@mail.muni.cz

***Abstract*—Formal methods are becoming an important tool for ensuring correctness and security of cryptographic constructions. However, the support for certain advanced proof techniques, namely rewinding, is scarce among existing verification frameworks, which hinders their application to complex schemes such as multi-party signatures and zero-knowledge proofs.**

We expand the support for rewinding in EasyCrypt by implementing a version of the general forking lemma by Bellare and Neven. We demonstrate its usability by proving EUF-CMA security of Schnorr signatures.

***Index Terms*—computer-aided cryptography, formal verification, EasyCrypt, rewinding, forking, Schnorr**

I. Introduction

Although provable security has gradually become a de facto standard in cryptography, it is long known that it is not a silver bullet that can rule out all attacks [15]. One of the leading problems is that security proofs are difficult to read and verify. As a result, they do not receive sufficient attention during the review process and subtle errors may stay unnoticed.

Several methods of structuring security proofs, such as the game-based technique [20], have been developed to reduce the effort required to both create and check such proofs, yet human verification will always be error-prone. This idea led to the development of *computer-aided cryptography*, which strives to apply formal verification techniques to security [6].

Despite the recent successes in this field — e.g., fixing and mechanizing security proofs for new post-quantum signature schemes, Dilithium [4] and XMSS [5] — there remain some classical results that seem to evade formal verification. Unforgeability of Schnorr signatures is one of them. This contrasts with the renewed interest in this construction and the number of novel Schnorr-based schemes (such as EdDSA, MuSig [18], and FROST [17]), particularly in the active area (as witnessed by NIST's call for proposals [9]) of multi-party threshold cryptography.

The likely reason for this discrepancy is that security reductions for Schnorr-like schemes commonly rely on *forking* of the hypothetical adversary — a special case of *rewinding* which is a technique that was not implemented by any cryptographic verification framework up until recently [12].

In this work, we build on the recent advancements in formalization of rewinding and address the described gap by implementing a *general forking lemma* in EasyCrypt and reproducing the well-known result on the security of simple Schnorr signatures. By doing so, we hope to pave the way for applying formal methods to more advanced constructions that previously appeared beyond reach — incl. some threshold schemes and zero-knowledge proofs (see Section V-B for more details). All EasyCrypt scripts are publicly available on GitHub[1].

II. Related work

Zain formally verified three signature schemes related to Schnorr using EasyCrypt [22] — EDL, CM, and GJKW. As the author explains, one of the reasons behind this choice was that all constructions feature a tight security reduction and do not require the forking lemma.

Several groups of authors formalized properties of zero-knowledge Σ-protocols and applied the results to the *interactive* Schnorr identification protocol; various verification frameworks were employed, e.g., CryptHOL [10], CertiCrypt [7], or EasyCrypt [13]. As far as we know, none of these works covered the properties of *non-interactive* protocols, or signature schemes obtained from interactive ones via the Fiat-Shamir transformation.

The rewinding technique was first implemented in EasyCrypt by Firsov and Unruh [12]. As this is the starting point of our work, we comment on it in more detail in Section III-B.

A partially overlapping work on forking is being carried out by Tuma and Hopper [21]. They develop an entirely new cryptographic verification framework, called *VCVio*, which they embed in the Lean programming language / theorem prover. Its focus is on design-level computational security (see the survey of computer-aided cryptography [6]) with emphasis on reasoning about oracles. To demonstrate its power, the authors use it to prove a variant of the forking lemma and show unforgeability of signatures constructed by the Fiat-Shamir heuristic.

To compare our work with that of Tuma and Hopper, we first note that our goal is to incorporate the forking technique into an established framework, EasyCrypt, instead of creating a new one. While the defining features of VCVio could prove useful in some settings, at the moment, EasyCrypt undoubtedly has an advantage in terms of the number of generic modular definitions and results available, which makes it easier to apply to new schemes. Moreover, EasyCrypt's Hoare logics (discussed in Section III-A) seem to be a key ingredient

[1] https://github.com/jjanku/fsec

for taming more complex proofs; given the novelty, it is again unclear how VCVio will fare in this direction without them. Also importantly, existing infrastructure allows us to transfer security guarantees from EasyCrypt onto an optimized implementation in Jasmin (see [3] and Section V-B); VCVio lacks a similar feature.

Regarding finer details, we observe that both our forking lemma and that in VCVio are loosely based on the hand-written general forking lemma by Bellare and Neven [8]. However, our result fully reproduces the probability bound from the original work and is better roughly by a factor of Q, the number of random oracle queries the forked algorithm makes. This stems from the fact that the authors of VCVio guess the forking point before running the algorithm for the first time, whereas we determine the forking point based on the result of the first execution.

Finally, we apply the forking lemma directly to the Schnorr signature scheme while Tuma and Hopper analyze the generic Fiat-Shamir transformation of a Σ-protocol with the necessary properties. This difference seems insignificant as the proofs are very similar and each work could be easily extended to cover the missing part.

To the best of our knowledge, there is no other formalization of the forking lemma and the security analysis of Schnorr signatures (in EasyCrypt or in other verification frameworks).

III. Preliminaries

First, we introduce the basics of EasyCrypt. Then, we recall the rewinding technique and summarize the existing formalization this work builds upon.

A. EasyCrypt

EasyCrypt is an interactive proof assistant tailored for formal verification of cryptographic proofs. It consists of three main components: a general-purpose *expression language*, a *module language* for specifying security games, and a set of *logics* for reasoning about them. We briefly illustrate the key concepts of each component on the following code snippet:

```
type log_t = bool list.

op dbool : bool distr.

module type Adv = {
  proc guess() : bool
}.

module B : Adv = {
  var log : log_t

  proc guess() : bool = {
    var b : bool;
    b ←$ {0,1}; (* uniform *)
    log ← log || [b];
    return b;
  }
}.

module Game(A : Adv) = {
  var won : bool

  proc play() = {
    var b0, b1 : bool;
    b0 ←$ dbool;
    b1 ← A.guess();
    won ← b0 = b1;
  }
}.
```

a) Expression Language: This is a typed functional language. Types and operators are built-in (`bool`, `=`) or user-defined (`log_t`, `dbool`) and can be abstract (`dbool`) or concrete (`log_t`). From the built-in type constructors, we point out `distr` which creates a type of discrete probability (sub)distributions over the supplied type (`dbool` is a distribution over `bool`). Evaluation of all EasyCrypt expressions is guaranteed to terminate, operators thus correspond to total functions.

b) Module Language: Unlike the expression language, the module language is imperative. There are standard statements such as assignments, procedure calls, or `while` loops. Additionally, one can assign a value at random from a given distribution using `←$`. Statements must be grouped into procedures, which must belong to modules.

Modules can be standalone (`B`) or parametrized (`Game`) by an abstract module (`A`), typically an oracle or an adversary, of a specified type (`Adv`). Module types are akin to interfaces that declare the procedures that the module must implement. A trivial method of interface inheritance is available using the `include` keyword.

Apart from procedures, modules may include global variables. These are not private and may be accessed by other modules. Accordingly, we define the set `glob M` of global variables[2] of a module `M`, as the union of global variables declared inside `M` and the set of variables `M` may read / write during a call to one of its procedures (i.e., `glob Game(B)` contains `B.log`).

An execution of a module's procedure is defined relative to a *memory* m, which defines the contents of all module's global variables. The state of a variable in a memory can be queried using curly braces (e.g., `Game.won{`m`}` or `(glob B){`m`}`). To compare states across two memories 1 and 2, a shorthand is available (`={B.log}` is equivalent to `B.log{1} = B.log{2}`).

c) Logics: EasyCrypt features three basic logics to reason about module properties: *classical* Hoare Logic (HL), *probabilistic* HL (pHL), and probabilistic *relational* HL (pRHL). Judgments in the given logic are denoted by the keyword `hoare`, `phoare`, and `equiv` respectively and take the following common form: `procedure(s) : precondition` $\Longrightarrow$ `postcondition`. Here, the conditions may refer to global variables; moreover, the keywords `arg` and `res` reference arguments and return values of the procedure(s) in question. (For pRHL, a side / memory specifier, `{1}` or `{2}`, is necessary to determine the procedure execution we are interested in. For pHL, a probability bound is additionally needed.)

[2]Technically, this is a type.

To prove general mathematical facts (about operators) and connect judgments from the specialized logics (about modules), the so-called *ambient* logic is provided. Using it, we can state `axiom`s (which are assumed to be true) and prove `lemma`s. Both can be parametrized (incl. by memories).

Formulas in the ambient logic can also contain *probability expressions*: `Pr[procedure(args) @ m : event]`. The semantics is the probability of `event` when executing `procedure` with the provided `args` from initial memory *m*. In `event`, the `res` keyword is again permitted.

Note that for clarity, we slightly deviate from EasyCrypt's syntax. In particular, we omit some types and typecasts (`%r`), use ligatures (∧ instead of `/\`), replace some operators (← instead of `<@`), simplify memory notation (italicized *m* instead of `&m`), etc. Readers familiar with EasyCrypt should have no trouble mapping our notation to the original one.

More information on EasyCrypt can be found in the EasyCrypt reference manual [11]. All limitations of EasyCrypt that are discussed in this paper apply to version 2025.03 (the latest release available to us at the time of writing).

B. Rewinding

Rewinding is a common cryptographic proof technique that generally proceeds as follows:

1) run a given program / adversary $\mathcal{A}$ for some time (typically until it requires some input from the environment),
2) save $\mathcal{A}$'s state,
3) finish the execution of $\mathcal{A}$,
4) restore $\mathcal{A}$'s state,
5) run $\mathcal{A}$ again till completion
6) (optionally repeat steps 4-5).

The idea behind rewinding is that we can extract some useful information from the multiple related outputs of $\mathcal{A}$ or (eventually) obtain an output that is otherwise suitable for us. Importantly, since the adversary has no knowledge of being rewound, its success probability can be constant over all individual runs.

The above paradigm can be applied to the forking lemma as well. For this reason, we based our work on the existing EasyCrypt formalization of rewinding by Firsov and Unruh [12]. Its short description follows.

To start, the authors first had to define what it means for a program to be *rewindable*. In pen-and-paper proofs, this capability is almost never introduced formally. In EasyCrypt, however, it is necessary to use the module types to explicitly require that a given module provides a rewinding interface:

```
type state_t.

module type Rewindable = {
  proc getState() : state_t
  proc setState(st : state_t) : unit
}.
```

Moreover, one has to axiomatize that an implementation of the above interface behaves as the procedure names suggest. Formally, a `Rewindable` module `R` should satisfy the following properties:

1) there exists some injective function `f` from `glob R` to `state_t`;
2) the procedure `getState` terminates, has no side-effects, and returns `f gR` when invoked in state `gR : glob R`;
3) the procedure `setState` terminates and sets the state of `R` to `gR` when called with an argument `st` such that `st = f gR`.

In the rest of this paper, all `Rewindable` modules are expected to satisfy the listed conditions; we omit them for brevity in the formulation of our lemmas. We also note that this assumption does not weaken our results; see the discussion in the original work [12].

The final step is to fix the point at which the module `R` should be rewound. To model this, the authors additionally assume that the execution of `R` is divided into two procedures, `init` and `run`. This is necessary because in EasyCrypt procedures are always executed in one go.

Once the rewinding interface is specified, it is possible to straightforwardly translate the initial informal description of rewinding into an EasyCrypt module:

```
module Rewinder(R : Rewindable) = {
  (* Two runs of R, with rewinding. *)
  proc run2(i) = {
    var st, r0, r1, r2;
    r0 ← R.init(i);
    st ← R.getState();
    r1 ← R.run(r0);
    R.setState(st);
    r2 ← R.run(r0);
    return ((r0, r1), (r0, r2));
  }

  (* Single run of R. *)
  proc run(i) = {
    var r0, r1;
    r0 ← R.init(i);
    r1 ← R.run(r0);
    return (r0, r1);
  }
}.
```

Listing 1. Rewinder module

A key result of Firsov and Unruh about the `Rewinder` module that we will utilize in Section IV-D intuitively states the following: if `R` succeeds once (its output satisfies some predicate `P`), then it should also succeed the second time after rewinding[3]. Formally:

```
lemma rewinding_lemma m P i :
  Pr[Rewinder(R).run2(i) @ m : P res.1 ∧ P res.2] ≥
  Pr[Rewinder(R).run(i)  @ m : P res] ^ 2
```

Listing 2. Rewinding lemma

The proof of this lemma relies on Jensen's inequality and *probabilistic reflection*, the ability to reason about denotational semantics of EasyCrypt modules [12].

[3]This result is less trivial than it might seem at first. Notice that the right-hand side of the inequality talks about probability of a full execution (i.e., initialization followed by the `run` procedure) while the probability on the left-hand side is with respect to a single initialization and two independent runs from the initialized state.

IV. Forking Lemma

In this section, we start by presenting the pen-and-paper formulation of the general forking lemma by which this work is inspired. In the following subsections, we describe our formalization and notable differences compared to the original work. At the end, we show how we specialize the general result for the common use-case with a random oracle.

A. Pen-and-Paper Formulation

The original forking lemma was formulated by Pointcheval and Stern [19]. However, we have decided to focus on a later variant of the lemma, the General Forking Lemma by Bellare and Neven [8]. The reason for this choice is two-fold. Firstly, the original statement deals specifically with digital signatures in the random oracle model. By opting for the generalized lemma, we hope to broaden the applicability of our work, potentially beyond signatures. Secondly, the modular nature of the latter lemma makes it arguably more amenable to formal verification.

The essence of the forking lemma, which Bellare and Neven capture in the generalized version, can be described as follows. We consider an algorithm $\mathcal{A}$ that solves a certain problem. Importantly, it receives from the environment a sequence of values sampled at random — whether as a result of interactions with an oracle or as part of its input — and the solution it outputs (if any) is based on one of the obtained values, say at index j. The goal is to obtain from $\mathcal{A}$ two solutions that would be *related* but based on two *distinct* values. To achieve it, we run $\mathcal{A}$ once, generate part of the sequence starting at index j anew (preserving the rest) and run $\mathcal{A}$ for a second time, hoping $\mathcal{A}$ derives its second solution again from the j-th value which differs from the first one. The lemma then bounds the probability that we succeed:

Lemma 1 (General Forking Lemma [8])**.** *Fix an integer $q \geq 1$ and let $\mathcal{A}$ be a randomized algorithm that takes an input $i \in I$ together with a sequence of values $h_0, \ldots, h_{q-1} \in H$, uses random coins $\omega \in \Omega$, and returns a number j from the range $0 \leq j < q$ on success or -1 on error and an auxiliary output $a \in A$.*

Next, consider the algorithm below:

$\mathsf{Forker}_{\mathcal{A}}(i)$

$\omega \leftarrow\!\!\$\ \Omega$ // *random coins*
$h_0, \ldots, h_{q-1} \leftarrow\!\!\$\ H$ // *oracle responses*
$(j, a) \leftarrow \mathcal{A}(i, h_0, \ldots, h_{q-1};\ \omega)$
if $j = -1$ **then**
  return $(-1, \epsilon, \epsilon)$
$h'_j, \ldots, h'_{q-1} \leftarrow\!\!\$\ H$
$(j', a') \leftarrow \mathcal{A}(i, h_0, \ldots, h_{j-1}, h'_j, \ldots, h'_{q-1};\ \omega)$
if $j = j' \wedge h_j \neq h'_j$ **then**
  return (j, a, a')
else
  return $(-1, \epsilon, \epsilon)$

Finally, for an input $i \in I$, let us define the accepting probability $acc(i)$ of $\mathcal{A}$ and the success probability $frk(i)$ of $\mathsf{Forker}_{\mathcal{A}}$:

$$acc(i) = \Pr[0 \leq j < q : (j, a) \leftarrow\!\!\$\ \mathcal{A}(i, h_0, \ldots, h_{q-1};\ \omega)],$$
$$frk(i) = \Pr[0 \leq j < q : (j, a, a') \leftarrow\!\!\$\ \mathsf{Fork}_{\mathcal{A}}(i)];$$

where in the former, the probability is taken over the choice of $h_0, \ldots, h_{q-1}$ and ω.

Then for every input $i \in I$, we have:

$$frk(i) \geq acc(i) \cdot \left(\frac{acc(i)}{q} - \frac{1}{|H|} \right).$$

Readers familiar with the original work may notice that an important part of the statement, averaging over different inputs i, is omitted here. For presentation purposes, we address this issue later in Section IV-F.

B. Adversary Model

One may observe that the algorithm $\mathsf{Forker}_{\mathcal{A}}$ does not quite match the definition of rewinding that we gave earlier in Section III-B — indeed, the state of $\mathcal{A}$ is never saved (nor restored). Instead, the algorithm $\mathcal{A}$ is run twice from the beginning on related inputs and with the same random tape ω.

To unify the two views on rewinding, suppose that $\mathcal{A}$ uses the provided values $h_0, \ldots, h_{q-1}$ one by one (this is also the typical setting). In that case, since the inputs i, the first j values h, and the random tapes are equal, the second execution of $\mathcal{A}$ can diverge from the first one only after $\mathcal{A}$ consumes the value h'_j. Therefore, running $\mathcal{A}$ as $\mathsf{Forker}_{\mathcal{A}}$ does is equivalent to rewinding $\mathcal{A}$, in the sense of Section III-B, to the state before it asks for h_j.

Although it should be possible to formalize both approaches to forking using EasyCrypt[4], we argue that the state-saving model is considerably more practical than the other model that fixes randomness. This is because EasyCrypt's *module language* is inherently stateful and probabilistic. For example, we cannot declare a deterministic abstract module.

One solution could be to limit ourselves to the functional *expression language* and model the rewound programs as operators (`op`; a total function). However, by that, we would give up on a significant proving power of EasyCrypt, such as the relational Hoare logic. Alternatively, we could let users of our forking theory prove that the provided modules are deterministic. But this poses a significant burden and is currently again a use-case out of the scope of EasyCrypt.

Moreover, both discussed workarounds would likely lead to loss of generality as distributions in EasyCrypt are discrete and pre-sampling the random coins for $\mathcal{A}$ would therefore force us to bound the amount of randomness $\mathcal{A}$ can use. Consequently, some almost-surely terminating programs (e.g., relying on rejection sampling), would be inexpressible in this model.

For the reasons above, we decided to further explore the state-saving model. This introduces a new challenge — how

[4]Though there may be further obstacles that we are currently unaware of.

do we save the state at the right time? The method outlined in Section III-B cannot record the state of a module mid a procedure call, it merely serializes the global variables of a module (not local). Hence if we wish to obtain the state of $\mathcal{A}$ before it consumes the j-th input value h, we must decompose the computation of $\mathcal{A}$ such that it yields control back to the caller before accessing h_j — similarly to how we split the computation of `module R` in Section III-B into procedures `init` and `run`. But since we learn the value of j only after $\mathcal{A}$ finishes, we require $\mathcal{A}$ to stop before processing *each* next value h. This leads us to the following interface for $\mathcal{A}$:

```
type query_t, resp_t.

type in_t, out_t.

module type Stoppable = {
  proc init(i : in_t) : query_t
  proc continue(r : resp_t) : query_t
  proc finish(r : resp_t) : out_t
}.
```

To mentally map $\mathcal{A}$ to `Stoppable`, think of `query_t` as the unit type, `resp_t` as H, `in_t` as I, and `out_t` as $(\{0,\ldots,q-1\}\cup\{-1\})\times A$.

To generate the responses (values of type `resp_t`) for a `Stoppable` module, we will use a module of the type below (the concrete module will be listed in a later section):

```
module type Oracle = {
  proc get(q : query_t) : resp_t
}.
```

The computation of any `Stoppable` module can then be naturally expressed using a `Runner` module:

```
(* Number of queries. *)
const Q : {int | 1 ≤ Q} as Q_pos.

module Runner(S : Stoppable, O : Oracle) = {
  proc run(i : in_t) : out_t = {
    var o : out_t;
    var q : query_t;
    var r : resp_t;
    var c : int;

    q ← S.init(i);
    c ← 1;

    while (c < Q) {
      r ← O.get(q);
      q ← S.continue(r);
      c ← c + 1;
    }

    r ← O.get(q);
    o ← S.finish(r);

    return o;
  }
}.
```

It can be shown by structural induction that any EasyCrypt module (with a `run` procedure) making a bounded number of oracle queries can be equivalently written as a composition of some `Stoppable` module and the `Runner` module.

We also wish to emphasize here that since the `Runner` module expresses the decomposed computation of a `Stoppable` module as a single procedure, it enables us to use the `Stoppable` module as an ordinary module; in particular, we do not have to modify the definitions of standard security notions in any way, as we shall see in Section V-A.

To conclude the discussion on the two different algorithm models, we can observe that both imply certain restrictions to the user in EasyCrypt. Despite that, we believe that the model we chose is superior to the other one, as it imposes only a syntactic restriction (implemented interface) as opposed to a semantic one (determinism and extracted randomness).

C. Forking Module

With the issue of rewinding being resolved in the previous section, we are now ready to formalize the forking algorithm in EasyCrypt.

We begin by specifying which modules can be forked. A `Forkable` module must implement the `Rewindable` as well as `Stoppable` interface so that its state can be queried at the appropriate times:

```
module type Forkable = {
  include Rewindable
  include Stoppable
}.
```

Here, the `Stoppable` type is a clone of the one defined in Section IV-B where `type out_t = int * aux_t` (cf. the output of $\mathcal{A}$ in Lemma 1).

We will consider an execution of a `Forkable` module successful iff the index it outputs is in range:

```
op success (j : int) : bool = 0 ≤ j < Q.
```

As alluded to before, we also have to define the response generator, or the oracle, for the `Stoppable` module. Following the General Forking Lemma 1 which generates values h independently of $\mathcal{A}$, we use a module that disregards the output (query) of the `Stoppable` module and always samples a fresh response. We call this oracle a *Forgetful Random Oracle* (FRO):

```
op [lossless uniform] dresp : resp_t distr.

module FRO : Oracle = {
  proc get(q : query_t) : resp_t = {
    var r : resp_t;
    r ←$ dresp;
    return r;
  }
}.
```

At this point, the definition of the forking module is straightforward, the main structure copies that of $\mathsf{Forker}_{\mathcal{A}}$:

```
module Forker(F : Forkable) = {
  var j1, j2 : int
  var log1, log2 : log_t list
  var r1, r2 : resp_t

  (* First full run of F. *)
  proc fst(i : in_t) : out_t * (log_t list) *
    (state_t list) = {
    (* ... *)
    c ← 1;
    while (c < Q) {
```

```
      st ← F.getState();
      sts ← sts ‖ [st];
      r ← Log(FRO).get(q);
      q ← F.continue(r);
      c ← c + 1;
    }
    (* ... *)
  }

  (* Second partial run of F. *)
  proc snd(q : query_t, c : int) : out_t * (log_t
    list) = {
    (* ... *)
    while (c < Q) {
      (* ... *)
      c ← c + 1;
    }
    (* ... *)
  }

  proc run(i : in_t) : int * aux_t * aux_t = {
    var sts : state_t list;
    var st : state_t;
    var j : int;
    var a1, a2 : aux_t;
    var q : query_t;

    ((j1, a1), log1, sts) ← fst();
    (q, r1) ← nth witness log1 j1;

    (* Rewind. *)
    st ← nth witness sts j1;
    F.setState(st);

    ((j2, a2), log2) ← snd(q, j1 + 1);
    log2 ← (take j1 log1) ‖ log2;
    r2 ← (nth witness log2 j1).2;

    j ← if success j1 ∧ j1 = j2 ∧ r1 ≠ r2
      then j1 else -1;

    return (j, a1, a2);
  }
}.
```

In the above, the procedure `fst` is a slight modification of `Runner(F, FRO).run` where we record the states `F` goes through and the responses it receives during its execution.

The procedure `snd` runs the module `F` as well with the difference that it responds to `F` just `Q - j` times, i.e., it can finish the execution of `F` after it is rewound to the `j`-th query.

D. Probability Analysis

For the forking algorithm presented in the preceding section, we prove a direct equivalent of the probability bound from Lemma 1:

```
const pr_collision =
  1 / (size (to_seq (support dresp))).

lemma pr_fork_success m i :
  let pr_runner_succ =
    Pr[Runner(F, FRO).run(i) @ m : success res.1] in
  let pr_fork_succ   =
    Pr[Forker(F).run(i) @ m : success res.1] in
  pr_fork_succ ≥ pr_runner_succ ^ 2 / Q -
    pr_runner_succ * pr_collision.
```

Proof Sketch. In spite of the change in the algorithm model, our formal proof of the forking lemma rather closely copies the original pen-and-paper proof [8]. The main distinction is that we invoke the Rewinding lemma (see Listing 2) at an appropriate time instead of proving an intermediate goal anew. We therefore only outline the main steps of the proof.

First, we observe that conditioned on `success j1`, the success event of the `Forker` is a conjunction of `j1 = j2` and `r1 ≠ r2`. We can thus bound the `Forker`'s success probability as shown below:

```
pr_fork_success ≥
  Pr[Forker(F).run(i) @ m :
     success Forker.j1 ∧ Forker.j1 = Forker.j2] -
  Pr[Forker(F).run(i) @ m :
     success Forker.j1 ∧ Forker.r1 = Forker.r2]
```

The latter probability in the difference is easy to analyze and leads to the term `pr_runner_succ * pr_collision` in the lemma. We perform a case analysis of the former probability:

```
Pr[Forker(F).run(i) @ m :
   success Forker.j1 ∧ Forker.j1 = Forker.j2] =
bigi predT (fun j ⇒
  Pr[Forker(F).run(i) @ m :
     Forker.j1 = j ∧ Forker.j2 = j]
) 0 Q
```

At this point, we can focus on `Pr[Forker(F).run(i) @ m : Forker.j1 = j ∧ Forker.j2 = j]` for a *fixed* `j`. A crucial step of our proof is showing that this probability remains the same if we modify `Forker` so that it always rewinds `F` to the `j`-th query:

```
module SplitForker(F : Forkable) = {

  (* Forker.fst that runs F only until the
   * first C queries. *)
  proc fst_partial(C : int) : query_t * (log_t
    list) * (state_t list) = {
    (* ... *)
  }

  (* Forker.snd, but with state recording. *)
  proc snd(q : query_t, c : int) : out_t * (log_t
    list) * (state_t list) = {
    (* ... *)
  }

  proc fst(C : int) : out_t * (log_t list) *
    (state_t list) = {
    (* ... *)
    (q, log1, sts1) ← fst_partial(C);
    (o, log2, sts2) ← snd(q, C);
    (* ... *)
  }

  proc run(i : in_t, j : int) : int * int * aux_t *
    aux_t * (log_t list) * (log_t list) = {
    (* ... *)
    ((j1, a1), log1, sts1) ← fst(j + 1);
    (* ... *)
    (* Rewind to the j-th query. *)
    q ← (nth witness log1 j).1;
    st ← nth witness sts1 j;
    F.setState(st);

    ((j2, a2), log2, sts2) ← snd(q, j + 1);
    (* ... *)
  }
}.
```

Notice that after this change, the `SplitForker(F).run` procedure roughly corresponds to `Rewinder(R).run2` (see Listing 1) if we consider `fst_partial` and `snd` to be the `init` and `run` procedures of the module `R` (respectively). This allows us, after some additional refactoring, to apply the Rewinding Lemma (Listing 2) and eventually obtain the inequality below:

```
Pr[Forker(F).run(i) @ m :
   Forker.j1 = j ∧ Forker.j2 = j] ≥
Pr[Runner(F, FRO).run(i) @ m : res.1 = j] ^ 2
```

The final step is to undo the case analysis and express the bound on the sum of probabilities in terms of a single event again. For that, we use the following lemma:

```
lemma square_sum (n : int) (f : int → real) :
  (1 ≤ n) ⇒
  (forall j, 0 ≤ j < n ⇒ 0 ≤ f j) ⇒
  bigi predT (fun j ⇒ square (f j)) 0 n ≥
    square (bigi predT f 0 n) / n.
```

The proof is an easy consequence of Jensen's inequality, which is present in EasyCrypt's standard `Distr` theory[5]. □

E. Property Transfer

In a pen-and-paper setting, the previous section could likely conclude our discussion of the forking lemma. Here, we need to do additional work to make the result practically usable. The problem is that the probability analysis of the success event alone gives us no information about the side outputs that are produced.

In a typical use-case, we would reason the following way: "If the program `F` succeeds, its side output `a` satisfies some property `P`; hence if the forker succeeds, both runs of `F` must have been successful and thus both auxiliary outputs `a1` and `a2` satisfy `P`." However, in our formal setting, this is not trivial as the two runs of `F` are essentially interleaved (due to the chosen rewinding approach). We therefore provide the `property_transfer` lemma that captures the implication above:

```
pred P_in : in_t * glob F.
pred P_out : out_t * (log_t list).

axiom run_prop : hoare[
  Runner(F, Log(FRO)).run :
  P_in (i, glob F) ∧ Log.log = [] ⟹
  P_out (res, Log.log)
].

hoare property_transfer :
  Forker(F).run :
  P_in (i, glob F) ⟹
  let (j, a1, a2) = res in success j ⇒
    P_out ((j, a1), Forker.log1) ∧
    P_out ((j, a2), Forker.log2).
```

Moreover, note that the predicate `P_out` is defined over `out_t * (log_t list)` so that we can express properties of the output with respect to the oracle log. Since the logs from the two executions share a prefix (see `Forker` in Section IV-C), this also allows us to relate the two side outputs — thereby replicate the reasoning that "because the values must have been computed before the forking point, they must be equal in both runs". This will prove useful later in Section V-A.

Proof Sketch. Showing that the output from the first run of `F` satisfies `P_out` is trivial. To show the same about the second output, we would, intuitively, want to use the following equivalence:

```
equiv[
  <skip> ~ st ← F.getState(); ...; F.setState(st) :
  ={glob F} ⟹ ={glob F}
].
```

The challenge is that we do not a priori know which `getState` call is relevant as we record the state multiple times and choose one only after we obtain the output index `j1`.

To circumvent this problem, we again rely on case analysis and the `SplitForker` module. The proof thus heavily utilizes the intermediate results of Section IV-D. □

For convenience, we also provide a `phoare` judgment that combines the probability analysis with the property transfer and some simple assertions about the two produced oracle logs; see the full code for details.

F. Averaging over Inputs

Many security games are played by first randomly generating some parameter (say a key) and only then letting the adversary attack. As a consequence, if we try to analyze the success probability of a reduction involving the `Forker` in such a game, the `pr_fork_success` lemma alone (Section IV-D) does not suffice — it is concerned with a fixed input whereas here, we need to average over all possibly generated inputs.

To enable application of the forking lemma in the described scenario, we first introduce a module type for input generators:

```
module type IGen = {
  proc gen() : in_t
}.
```

And then, we create modified versions of the `Forker` and `Runner` modules presented earlier:

```
module IForker(I : IGen, F : Forkable) = {
  (* ... *)
  proc fst() : out_t * (log_t list) * (state_t
    list) = {
    (* ... *)
    i ← I.gen();
    q ← F.init(i);
    (* run F *)
  }
  (* ... *)
}.

module IRunner(I : IGen, S : Stoppable, O : Oracle)
    = {
  proc run() : out_t = {
    var i, o;
    i ← I.gen();
```

[5] https://github.com/EasyCrypt/easycrypt/blob/r2025.03/theories/distributions/Distr.ec

```
    o ← Runner(S, O).run(i);
    return o;
  }
}.
```

For these modules, we prove analogous lemmas as those presented in Sections IV-D and IV-E (we do not list them again for brevity).

Proof Sketch. While Bellare and Neven first prove the probability bound for a fixed input and then solve the average case via Jensen's inequality and basic properties of expectation, we chose a different approach here.

We notice that the Rewinding Lemma (Section III-B, [12]) already involves some averaging, namely over all the different states the rewound module may end in after running the `init` procedure. The proof in Section IV-D therefore goes through with almost no modifications needed even if we add the `I.gen()` call as shown.

Practically, this has the implication that we prove all lemmas directly for `IForker` and recover the earlier results by using a constant input generator. □

G. Random Oracle Model

Although the flexibility of the forking lemma in its general form can be essential to handle more complex cases, we also implement a specialized version of the lemma for the arguably more common random oracle setting.

Specifically, we add a new `IForkerRO` module that provides the executed module with a standard *Lazy Random Oracle* (LRO) — unlike the earlier `IForker` which provides a *Forgetful Random Oracle* (FRO) — and prove the usual probability bounds.

One notable accommodation we have to make for this setup is to change the output value type of the forked module. It no longer makes sense to return an *index* of a query as repeated queries need to return the same response, i.e., we cannot freely generate a new one after the module is rewound. Instead, we require the forked module to return some queried *value*; `IForkerRO` then rewinds the module to the point where the value was first queried. This change is reflected in the definition of the `ForkableRO` module type.

Proof Sketch. Internally, `IForkerRO` executes `IForker` parametrized by the module `F : ForkableRO` wrapped in `Red_LRO_FRO`. This module creates an overlay above the `FRO` oracle that simulates `LRO`.

Technically, this is achieved by copying *all* `F`'s queries, recording the responses of `FRO`, and fixing those that would create an inconsistency in the simulation before passing them to `F`[6]:

```
module Red_LRO_FRO(F : ForkableRO) : Forkable = {
  var q : query_t
  var m : log_t list

  (* proc getState(), setState(), finish() *)

  proc init(i : in_t) : query_t = {
    m ← [];
    q ← F.init(i);
    return q;
  }

  proc fix_resp(r : resp_t) : resp_t = {
    m ← m ‖ [(q, r)];
    (* Return 1st response associated with q. *)
    r ← oget (assoc m q);
    return r;
  }

  proc continue(r : resp_t) : query_t = {
    r ← fix_resp(r);
    q ← F.continue(r);

    return q;
  }
}.
```

A key step is to show that the above simulation is indeed correct, formally:

```
equiv red_log_fro_lro_equiv :
  IRunner(I, Red_LRO_FRO(F), Log(FRO)).run ~
    IRunnerRO(I, F, LRO).run :
  ={glob I, glob F} ∧ Log.log{1} = [] ∧ LRO.m{2} =
    empty ⟹
  ={glob I, glob F} ∧ ofassoc Log.log{1} = LRO.m{2}.
```

The rest of the proof involves a simple application of the general forking lemma. □

V. Applications

To show that our formalization of the forking lemma is practical, we apply it in the security analysis of Schnorr signatures. We also discuss possible future applications.

A. Schnorr Signatures

The Schnorr scheme is a digital signature scheme derived from the interactive Schnorr identification scheme via the Fiat-Shamir transformation. Since it was among the first examples of usage of the original forking lemma by Pointcheval and Stern [19] and it is still highly relevant today, it presents a natural first test for our forking lemma.

We start by defining an EasyCrypt module, called `Schnorr`, corresponding to the scheme in the random oracle (RO) model. The definition is given for an arbitrary cyclic group of prime order `order` and the RO responses are drawn from a distribution `dchal`. As it is otherwise a direct copy of the well-known pseudocode, we do not list it here and refer the reader to the Appendix instead (Listing 3).

In the security analysis, we begin with existential unforgeability against key-only attacks (EUF-KOA, [14]) and move on to security against chosen message attacks (EUF-CMA, [14]) next; both results are relative to the hardness of the discrete logarithm (DL) problem. All mentioned notions are formalized in EasyCrypt's standard library

[6]Note that we could have also copied only `F`'s *unique* queries, i.e., return from the `continue` function only when `F` makes a new query. This approach appears to be more challenging as it leads to nested `while` loops once the `Runner` module is involved.

(in the DigitalSignaturesROM[7] and DLog[8] theories, respectively) and we use them without any change[9]. Our adversary model includes *forkable* attackers only (in the sense of Section IV).

Let us now proceed to the formulation of the first lemma, which informally says that if an attacker A making at most QR random oracle queries has non-negligible probability of forging a signature in the EUF_KOA_ROM game, then the attacker Red_KOA_DL(A) has non-negligible probability of solving the DL problem in the Exp_DL experiment, assuming support dchal is large enough:

```
lemma schnorr_koa_secure m :
  Pr[Exp_DL(Red_KOA_DL(A)).main() @ m : res] ≥
    Pr[EUF_KOA_ROM(LRO, Schnorr,
       FAdv_KOA_Runner(A)).main() @ m : res] ^ 2
    / (QR + 1) -
    Pr[EUF_KOA_ROM(LRO, Schnorr,
       FAdv_KOA_Runner(A)).main() @ m : res]
    / (size (to_seq (support dchal))).
```

Note that to be able to use A in the EUF_KOA_ROM game which expects an adversary implementing the forge procedure, we wrap it in the FAdv_KOA_Runner module where forge = Runner(A).run.

Proof Sketch. The formal proof copies the classical argument. First, we build a new Forkable module AdvWrapper(A) that runs A, verifies the returned forgery, and, if valid, outputs the *critical query* related to it (along with the forgery itself as a side output). From the definition, it follows that the probability AdvWrapper(A) returns some query is equal to the probability that A wins the EUF_KOA_ROM game.

Next, we define the DL adversary Red_KOA_DL(A) which utilizes ForkerRO to rewind AdvWrapper(A) to the critical query and (hopefully) obtain two related valid signatures from it. These then suffice to extract the private key because the proof system underlying Schnorr signatures satisfies *special soundness*.

Finally, we apply the forking lemma to establish the probability bound on the success of the DL adversary. Here, we wish to highlight how the property transfer from Section IV-E crucially simplifies the proof. We show that if AdvWrapper(A) succeeds, then the returned forgery is valid w.r.t. the produced random oracle log. Property transfer then gives us that if ForkerRO succeeds, both forgeries are valid, each w.r.t. its corresponding log. Moreover, from the forking lemma, we know that these logs share a prefix. Together, this implies that the two forged signatures are related in a way satisfying the preconditions of the special soundness extractor. In summary, the property transfer allows us to focus on a *single run* of AdvWrapper(A) and treat the forker as a *black box*. □

As we outlined earlier, our second result deals with EUF-CMA security. This implies that the adversary A is now additionally allowed to make up to QS signature queries to the signature oracle BoundedSO which stops responding after this limit is reached. It is also worth mentioning how we provide A with this oracle; we define the adversary type below:

```
module type FAdv_CMA (SO : SOracle_CMA_ROM) = {
  include Stoppable
  include Rewindable
}.
```

As can be seen, we combine the Stoppable interface with EasyCrypt's built-in method of oracle access, the two approaches can be mixed and do not necessarily clash with each other.

```
lemma schnorr_cma_secure m :
  let pr_cma_succ =
    Pr[EUF_CMA_ROM(LRO, Schnorr, FAdv_CMA_Runner(A),
       BoundedSO).main() @ m : res] in
  let pr_dl_succ =
    Pr[Exp_DL(Red_KOA_DL(Red_CMA_KOA(A))).main()
       @ m : res] in
  pr_cma_succ ≥ QS * (QS + QR) / order ⇒
  pr_dl_succ ≥
    (pr_cma_succ - QS * (QS + QR) / order) ^ 2 /
    (QR + 1) - 1 / (size (to_seq (support dchal))).
```

Proof Sketch. As the formulation of the lemma suggests, we reduce the EUF-CMA setting to the EUF-KOA case. This is in contrast to some pen-and-paper proofs, which reduce EUF-CMA directly to DL — our approach is more layered. It also implies that the proof is uninteresting from the point of view of forking as we have already applied the lemma of interest in the preceding proof. This proof does, however, further show how one can work with Stoppable modules.

We also note that the EUF-CMA-to-EUF-KOA reduction was formalized as part of a previous work on the Fiat-Shamir transformation with aborts [4]. Our problem is considerably simpler as the Schnorr identification scheme always terminates successfully (with honest parties).

The definition of the reduction can be found in the Red_CMA_KOA module which resembles the Red_LRO_FRO module seen in Section IV-G. We again build a certain *overlay* over the provided random oracle by replaying A's queries and modifying the received responses, if necessary, before passing them to A. This method enables us to gain control over the oracle and program it freely as required by the *honest verifier zero knowledge* simulator associated to the Schnorr protocol which we use to respond to A's signature queries.

It may, of course, happen that the overlay already contains an entry for the query whose response we need to program and the simulation fails as a result. We use the fel (Failure Event Lemma) tactic to bound the probability of this happening during a run of Red_CMA_KOA(A). Conveniently, this tactic can be applied to a sequence of statements (as opposed to a single call to an abstract module); having the execution of A span across multiple calls (init, continue in a loop, and finish) therefore poses no significant hurdle.

To conclude the proof, it remains to show that if A wins EUF_CMA_ROM, then Red_CMA_KOA(A) wins EUF_KOA_ROM (provided the bad event does not occur).

[7]https://github.com/EasyCrypt/easycrypt/blob/r2025.03/theories/crypto/DigitalSignaturesROM.eca

[8]https://github.com/EasyCrypt/easycrypt/blob/r2025.03/theories/crypto/DLog.ec

[9]With the exception of renaming DLogExperiment to Exp_DL.

Here, the main challenge is that in the former game, the forgery is verified against a RO which contains queries made by the signature oracle whereas in the latter game, the *overlaid* RO is used which may not be defined for these queries (unlike the *overlay*). Ultimately, this difference does not matter as a forgery for an already signed message is invalid. However, expressing the relationship between the RO and its overlay forms a substantial part of the formal proof. □

B. Future Work

Given the broad use of the forking lemma, there are multiple directions in which we could extend our current work.

The most obvious choice is to connect our results on the security of Schnorr signatures with a prior work on the *interactive* Schnorr protocol [2] and hash functions[10] in Jasmin [1], a framework designed for "high-assurance and high-speed cryptography". This way, by means of semantics-preserving extraction of Jasmin to EasyCrypt [3], we could create a fully verified implementation of the Schnorr signature scheme. In the same vein, we could also build on the ongoing Ed25519 Jasmin implementation effort[11].

Another option is to test the limits of our formalization on one of the Schnorr-based multi-party signature schemes from the recent series of works in this area. Some schemes apply the forking lemma in non-trivial ways (such as forking at two different points [18]), while others rely on various modifications of this lemma [16].

Finally, we could also continue with the formalization of security properties of zero-knowledge protocols [13] and generalize the statements from the previous section to the setting where the Fiat-Shamir transformation is applied to an arbitrary sigma protocol satisfying special soundness and honest verifier zero knowledge.

VI. Conclusion

This paper introduced the forking technique into EasyCrypt. We first defined an appropriate model for forkable algorithms. Next, we formalized a version of the general forking lemma. Finally, we showed how our result can be used to derive the standard bound on security of Schnorr signatures.

To our knowledge, no previous work focused on forking in EasyCrypt. Our work therefore crucially expands what is in reach of this popular verification framework and we hope it will serve as a stepping stone towards formal analysis of new, more involved cryptographic constructions — notably threshold signatures and zero-knowledge proofs.

[10] https://github.com/formosa-crypto/libjade

[11] https://github.com/formosa-crypto/formosa-25519/pull/30

References

[1] José Bacelar Almeida et al. "Jasmin: High-Assurance and High-Speed Cryptography". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 1807–1823. ISBN: 9781450349468. DOI: 10.1145/3133956.3134078. URL: https://doi.org/10.1145/3133956.3134078.

[2] José Bacelar Almeida et al. *Schnorr protocol in Jasmin*. Cryptology ePrint Archive, Paper 2023/752. 2023. URL: https://eprint.iacr.org/2023/752.

[3] José Bacelar Almeida et al. "The Last Mile: High-Assurance and High-Speed Cryptographic Implementations". In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020, pp. 965–982. DOI: 10.1109/SP40000.2020.00028.

[4] Manuel Barbosa et al. "Fixing and Mechanizing the Security Proof of Fiat-Shamir with Aborts and Dilithium". In: *Advances in Cryptology – CRYPTO 2023*. Ed. by Helena Handschuh and Anna Lysyanskaya. Cham: Springer Nature Switzerland, 2023, pp. 358–389. ISBN: 978-3-031-38554-4.

[5] Manuel Barbosa et al. "Machine-Checked Security for XMSS as in RFC 8391 and SPHINCS+". In: *Advances in Cryptology – CRYPTO 2023*. Ed. by Helena Handschuh and Anna Lysyanskaya. Cham: Springer Nature Switzerland, 2023, pp. 421–454. ISBN: 978-3-031-38554-4.

[6] Manuel Barbosa et al. "SoK: Computer-Aided Cryptography". In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021, pp. 777–795. DOI: 10.1109/SP40001.2021.00008.

[7] Gilles Barthe et al. "A Machine-Checked Formalization of Sigma-Protocols". In: *2010 23rd IEEE Computer Security Foundations Symposium*. 2010, pp. 246–260. DOI: 10.1109/CSF.2010.24.

[8] Mihir Bellare and Gregory Neven. "Multi-signatures in the plain public-Key model and a general forking lemma". In: *Proceedings of the 13th ACM Conference on Computer and Communications Security*. CCS '06. Alexandria, Virginia, USA: Association for Computing Machinery, 2006, pp. 390–399. ISBN: 1595935185. DOI: 10.1145/1180405.1180453. URL: https://doi.org/10.1145/1180405.1180453.

[9] Luís T. A. N. Brandão and René Peralta. *NIST First Call for Multi-Party Threshold Schemes*. Tech. rep. National Institute of Standards and Technology, 2023. DOI: 10.6028/NIST.IR.8214C.ipd.

[10] D. Butler et al. "Formalising Σ-Protocols and Commitment Schemes Using CryptHOL". In: *J. Autom. Reason.* 65.4 (Apr. 2021), pp. 521–567. ISSN: 0168-7433. DOI: 10.1007/s10817-020-09581-w. URL: https://doi.org/10.1007/s10817-020-09581-w.

[11] *EasyCrypt Reference Manual*. 2024. URL: https://www.easycrypt.info/easycrypt-doc/refman.pdf.

[12] Denis Firsov and Dominique Unruh. "Reflection, rewinding, and coin-toss in EasyCrypt". In: *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2022. Philadelphia, PA, USA: Association for Computing Machinery, 2022, pp. 166–179. ISBN: 9781450391825. DOI: 10.1145/3497775.3503693. URL: https://doi.org/10.1145/3497775.3503693.

[13] Denis Firsov and Dominique Unruh. " Zero-Knowledge in EasyCrypt ". In: *2023 IEEE 36th Computer Security Foundations Symposium (CSF)*. Los Alamitos, CA, USA: IEEE Computer Society, July 2023, pp. 1–16. DOI: 10.1109/CSF57540.2023.00015. URL: https://doi.ieeecomputersociety.org/10.1109/CSF57540.2023.00015.

[14] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. "A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks". In: *SIAM Journal on Computing* 17.2 (1988), pp. 281–308. DOI: 10.1137/0217017. eprint: https://doi.org/10.1137/0217017. URL: https://doi.org/10.1137/0217017.

[15] Neal Koblitz and Alfred Menezes. *Critical perspectives on provable security: Fifteen years of "another look" papers*. 2019. DOI: 10.3934/amc.2019034. URL: https://www.aimsciences.org/article/id/5d2d9acd-8f7a-4e46-8e4a-dfab9701f215.

[16] Chelsea Komlo. *A Note on Various Forking Lemmas*. 2023. URL: https://www.chelseakomlo.com/assets/content/notes/Forking-Lemma-Variants.pdf.

[17] Chelsea Komlo and Ian Goldberg. "FROST: Flexible Round-Optimized Schnorr Threshold Signatures". In: *Selected Areas in Cryptography*. Ed. by Orr Dunkelman, Michael J. Jacobson Jr., and Colin O'Flynn. Cham: Springer International Publishing, 2021, pp. 34–65. ISBN: 978-3-030-81652-0.

[18] Gregory Maxwell et al. "Simple Schnorr multi-signatures with applications to Bitcoin". In: *Des. Codes Cryptography* 87.9 (Sept. 2019), pp. 2139–2164. ISSN: 0925-1022. DOI: 10.1007/s10623-019-00608-x. URL: https://doi.org/10.1007/s10623-019-00608-x.

[19] David Pointcheval and Jacques Stern. "Security Proofs for Signature Schemes". In: *Advances in Cryptology — EUROCRYPT '96*. Ed. by Ueli Maurer. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 387–398. ISBN: 978-3-540-68339-1.

[20] Victor Shoup. *Sequences of games: a tool for taming complexity in security proofs*. Cryptology ePrint Archive, Paper 2004/332. 2004. URL: https://eprint.iacr.org/2004/332.

[21] Devon Tuma and Nicholas Hopper. *VCVio: A Formally Verified Forking Lemma and Fiat-Shamir Transform, via a Flexible and Expressive Oracle Representation*. Cryptology ePrint Archive, Paper 2024/1819. 2024. URL: https://eprint.iacr.org/2024/1819.

[22] Sara Zain. "Machine-checked verification of digital signature schemes in EasyCrypt". PhD thesis. University of Bristol, 2023. URL: https://hdl.handle.net/1983/0c762e35-d43e-4ce7-a7a6-780be08991d1.

APPENDIX

```
(* Import definiton of a cyclic "group" of prime
 * "order" with a generator "g" from EasyCrypt's
 * standard library. The type "exp" corresponds to
 * ZZ/"order". *)

type com_t  = group. (* Commitment *)
type chal_t = exp.   (* Challenge  *)
type resp_t = exp.   (* Response   *)
type trans_t =       (* Transcript *)
  com_t * chal_t * resp_t.

type pk_t = group.
type sk_t = exp.

type msg_t.
type sig_t = com_t * resp_t.

op [lossless uniform] dnonce : exp distr.
op [lossless uniform] dsk : sk_t distr.
op [lossless uniform] dchal : chal_t distr.

op verify (pk : pk_t) (t : trans_t) =
  let (com, chal, resp) = t in
  g ^ resp = com * (pk ^ chal).

module (Schnorr : Scheme_ROM) (RO : Oracle) = {
  proc keygen() : pk_t * sk_t = {
    var sk, pk;
    sk <$- dsk;
    pk <- g ^ sk;
    return (pk, sk);
  }

  proc sign(sk : sk_t, m : msg_t) : sig_t = {
    var pk, nonce, com, chal, resp;

    pk <- g ^ sk;
    nonce <$- dnonce;
    com <- g ^ nonce;
    chal <- RO.get(pk, com, m);
    resp <- nonce + sk * chal;

    return (com, resp);
  }

  proc verify(pk : pk_t, m : msg_t, s : sig_t) :
    bool = {
    var com, resp, chal;

    (com, resp) <- s;
    chal <- RO.get(pk, com, m);

    return verify pk (com, chal, resp);
  }
}.
```

Listing 3. Schnorr signature scheme