Zero-Knowledge 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*—We formalize security properties of zero-knowledge protocols and their proofs in EasyCrypt. Specifically, we focus on sigma-protocols (three-round protocols). Most importantly, we also cover properties whose security proofs require the use of rewinding; prior work has focused on properties that do not need this more advanced technique. On our way we give generic definitions of the main properties associated with sigma protocols, both in the computational and information-theoretical setting. We give generic derivations of soundness, (malicious-verifier) zero-knowledge, and proof of knowledge from simpler assumptions with proofs which rely on rewinding. Also, we address sequential composition of sigma protocols. Finally, we illustrate the applicability of our results on three zero-knowledge protocols: Fiat-Shamir (for quadratic residues), Schnorr (for discrete logarithms), and Blum (for Hamiltonian cycles, NP-complete).**

***Index Terms*—cryptography, formal methods, EasyCrypt, zero-knowledge, sigma protocols, rewinding**

Contents

1. Introduction

Zero-knowledge (ZK) protocols are cryptographic protocols that allow a prover to convince a verifier that they possess certain knowledge, without revealing that knowledge. More formally, let R be a relation. Then a ZK protocol for the relation R allows the prover to convince the verifier that the prover knows a witness w for some given statement s, so that $(s, w) \in R$, without revealing anything else about w itself. For example, the prover may prove that a hash s is a hash of some well-formed data w. (In which case R consists of all pairs (s, w) with $s = hash(w)$ and w well-formed.) ZK protocols constitute an important building block in cryptography as they can help to enforce the honest behaviour from potentially malicious parties. For example, the proof can provide a guarantee that the party is authorised to perform certain actions or access certain sensitive information.

The security of ZK protocols is expressed via properties of completeness, soundness, zero-knowledge, and proof of knowledge. Completeness ensures the correct operation of the protocol if both prover and verifier follow the protocol honestly. Soundness ensures that for "wrong" statements (i.e., with no witness) a prover can convince the verifier with only a small probability. Proof of knowledge guarantees that any prover that successfully convinces the verifier actually knows a witness (and not only abstractly that it exists). Zero-knowledge establishes that any cheating verifier cannot learn anything about the witness when running the protocol. These properties are typically shown by mathematical pen-and-paper proofs.

Pen-and-paper proofs are, however, 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 relationships between programs are analyzed in a probabilistic relational Hoare logic (pRHL). EasyCrypt has been successfully used to verify a variety of cryptographic schemes, such as electronic voting [2], digital signatures [3], differential privacy [4], security of IPsec [5], etc.

Properties related to ZK protocols are challenging to prove formally. For example, the proofs of the important zero-knowledge property requires a technique known as rewinding of adversaries. To the best of our knowledge, until recently rewinding was unavailable in EasyCrypt and other popular cryptography-oriented theorem provers. As a result, properties relying on rewinding were never properly addressed in formal setting. As we show in the related work section (Sec. 1.1), the existing formalization efforts of sigma-protocols mostly addressed properties which do not depend on rewinding which include completeness, special soundness, and honest-verifier zero-knowledge.

Recently the rewinding of programs was formally implemented in EasyCrypt [6].

This motivated us to generically formalize derivations of (malicious-verifier) zero-knowledge, proof of knowledge, and soundness in EasyCrypt. We address a specific but very common subclass of ZK protocols, namely sigma-protocols. These are three-message protocols of a certain specific structure (see Sec. 3.2). Our technical contributions include the following results:

- We give generic definitions of the main properties associated with sigma protocols. These include completeness, soundness, zero-knowledge, special soundness, and proof of knowledge (a.k.a. extractability). We address computational and information-theoretical versions of these properties (see Sec. 3).
- We present generic derivations of soundness from extractability, extractability from special soundness, and zero-knowledge from "one-shot" simulators (see Sec. 4).
- We prove the sequential compositionality for completeness, soundness, and zero-knowledge of sigma protocols (see Sec. 5).
- We instantiate our results for three ZK protocols: Fiat-Shamir[1] (for quadratic residues) [7], Schnorr (for discrete logarithms) [8], and Blum (for Hamiltonian cycles, NP-complete) [9] (see Sec. 6).

Our EasyCrypt formalization is available on GitHub [10]. There we provide instructions for running the code (file `README.md`) as well as instructions about the structure of the development and how use them for own developments (file `MANUAL.md`).

1.1. Related Work

In [11], Barthe et al. give one of the first machine-checked formalziation of sigma-protocols in CertiCrypt. Their main focus is on a subclass of sigma protocols that is aimed at proving knowledge of pre-images under group homomorphisms. This limits the applicability of their results to problems that exhibit this particular algebraic structure (e.g., this excludes Blum's protocol). In their work, the authors give a generic definition of one run of the sigma protocol (i.e., exchange of three messages) and formalize some of its properties, namely, completeness, special soundness, and honest-verifier zero-knowledge. They formalize the "perfect" variant of these properties (as compared to statistical and computational) and the AND/OR compositionality of sigma protocols. The applicability of their results is illustrated on examples which include Schnorr, Okamoto, Guillou-Quisquater, and Feige-Fiat-Shamir protocols. The authors do not address (malicious-verifier) zero-knowledge, proof of knowledge, soundness, and sequential composition of sigma-protocols.

Another related field is the development of verified cryptographic compilers. In the context of ZK protocols, important examples of these are the CACE compiler [12] and ZKCrypt [13].

The CACE compiler [12] is a certifying compiler that generates efficient implementations of zero-knowledge protocols. The CACE compiler takes abstract specifications of a zero-knowledge protocol and generates C or Java implementations. The main compilation steps are certifying in the sense that they generate an Isabelle proof of special soundness. However, the zero-knowledge, completeness, and soundness properties are not addressed.

Almeida et al. present ZKCrypt [13] which is an optimizing cryptographic compiler for sigma protocols. Similarly to the work by Barthe et al. [11], the authors consider

1. This should not be confused with Fiat-Shamir transformation.

only the class of sigma protocols for proving knowledge of pre-images under group homomorphisms. ZKCrypt implements two compilers: "verified" and "verifying". The verified compiler takes an abstract description of a sigma protocol and generates a reference implementation. The verifying compiler outputs an optimized implementation (in C or Java) which is provably equivalent to the reference implementation. Most importantly, the proofs returned by the compilers establish that the reference and optimized implementations satisfy the perfect completeness, special soundness, and honest-verifier zero-knowledge. The soundness, (malicious-verifier) zero-knowledge, proof of knowledge, and sequential composition are not addressed.

In [14], Butler et al. used the CryptHOL framework to formalize and derive commitment schemes from sigma protocols. The applicability of their work is illustrated by instantiating the Schnorr, Chaum-Pedersen, and Okamoto sigma protocols. The authors derive completeness, special-soundness, and honest-verifier zero-knowledge. The highlight of their work is a generic construction of commitment schemes from sigma protocols. In their work, the authors do not address (malicious-verifier) zero-knowledge, soundness, proof of knowledge, and sequential compositionality.

In [15], Alemeida et al. give a machine-checked implementation of a framework that allows users to construct efficient zero-knowledge protocols from secure multiparty computation protocols. For their generic constructions, the authors formalize the security definitions and proofs related to completeness, soundness, and zero-knowledge of sigma protocols. The authors do not address special soundness, proof of knowledge, and sequential compositionality of zero-knowledge. Their framework is implemented in EasyCrypt and some definitions are similar to ours, but there are also important differences.

In their formalization, the authors define the honest prover as a pure function (i.e., not as EasyCrypt procedure/module) which takes randomness neccessary for its computations as one of its arguments. This approach can drastically simplify the proofs, but makes the instatiation harder especially in cases when prover needs to use other cryptographic primitives such as commitments (in which case also these primitives must be modelled as pure functions with explicit randomness). The most significant overlap with our results is the derivation of malicious-verifier zero-knowledge from one-shot simulator which is similiar to our result in Sec. 4.1. However, the important difference is that for this result the authors change their representation of malicious verifiers and honest provers. Both, the honest prover and a malicious verifier, are now modelled semantically; in other words, these parties are not represented as stateful programs (i.e., not as EasyCrypt procedure/module), but as parameterized mathematical distributions instead. This approach greatly simplifies proofs (e.g., rewinding in this model is a trivial re-sampling), but makes it harder (if not impossible) to combine it with other definitions which rely on the standard representation of protocol parties as programs. Indeed, in other parts of their formalization, the authors work with standard representation which strictly speaking makes their own results incompatible with each other.

Also, they establish a security level for zero-knowledge which equals to $2\epsilon N + p^N$, where ϵ is a security bound for one-time simulator, p is the probability of a "bad"-event, and N is the number of tries performed by the simulator. Thus their bound becomes linearly worse in the number of tries. In contrast, we obtain the bound $\epsilon + p^N$ which approaches ϵ exponentially quickly in the number of tries.

In [16], Sidorenco et al. also perform a formal analysis of MPC-in-the-head zero-knowledge protocols in EasyCrypt. The authors provide a machine-checked security proof of a zero-knowledge protocol which follows the MPC-in-the-head paradigm. Their mechanization specifically studies the ZKBoo protocol [17]. They prove completeness, soundness, and honest-verifier zero-knowledge. Similarly to the work by Almeida et al. the work by Sidorenco et al. introduces basic zero-knowledge definitions of completenss and special soundness which are similar to ours. They do not address soundness, (malicious-verifier) zero-knowledge, proof of knowledge, and sequential composition of sigma protocols.

All these results focus on definitions and proofs that can be formalised without rewinding of adversaries (except [15] where adversaries are modelled as distributions). We believe the rewinding-related properties to be the current frontier in the mechanization of ZK proofs. In this work we set out to overcome this hurdle.

2. Preliminaries

2.1. Basics

In the following we comment on the main constructs of EasyCrypt which include types, operators, lemmas, axioms, theories, module types, and modules. More information on EasyCrypt can be found in the EasyCrypt tutorial [18].

Types and Operators. EasyCrypt has built-in and user defined types. The examples of built-in types are `bool`, `int`, `real`, and `unit`. Also, every type `t` is associated with a type `t distr` of its discrete (sub)probability distributions. A discrete (sub)probability distribution over a type is defined by its probability mass function, i.e. by a non-negative function from `t` to `real`. Also, EasyCrypt allows users to define recursive datatypes and functions based on a polymorphic typed lambda calculus. On the top-level, pure functions can be defined using the `op` keyword. For example, we can define the negation on booleans as follows:

```
op not (b: bool): bool =
  if b then false else true.
```

In addition, the standard library of EasyCrypt includes the implementation and properties of lists, arrays, finite sets, maps, probability distributions, etc.

Ambient Logic. EasyCrypt has built-in logics which are specialized for reasoning about programs (such as Hoare logic). Furthermore it implements an ambient logic which is a higher-order classical logic for proving mathematical facts and connecting judgements from the other logics. For example, we can use ambient logic to prove that double negation is identity. In lemmas and axioms we will use symbols ∀ and ∃ instead of official keywords `forall` and `exists`, respectively.

```
lemma notnot: ∀ (b: bool), not (not b) = b.
proof. progress. smt. qed.
```

In EasyCrypt, proofs consist of series of tactic applications which either discharge the proof obligation or transform it into new subgoal(s).

Theories. In EasyCrypt, theories can be used to group together related definitions. Theories can have parameters in the form of declared but undefined operators, types, and axioms. For example:

```
theory MonoidTheory.
  % parameters
  type M.

  op f: M → M → M.
  op e: M.

  axiom assoc: ∀ a b c, f (f a b) c = f a (f b c).
  axiom elaws: ∀ a, f a e = a ∧ f e a = a.

  % more useful results and definitions...
end MonoidTheory.
```

Later, the theory can be "cloned" and the operators and types instantiated with concrete values for which the axioms are provable. This enables modular design of theories.

Modules. In EasyCrypt, cryptographic games are modelled as modules, which consist of procedures written in a simple imperative language. Modules may be parameterized by abstract modules. Modules can be stateful, having global variables. The global variables of a module contains the variables declared in the module and any variables its procedures can access (directly or indirectly).

For example, we can implement a module `BitSampler` which has one procedure and one global variable `log`. The procedure `run` samples a uniform Boolean `b`, adds the result to the log, and returns `b` as the result of the call:

```
module BitSampler = {
  var log: bool list
  proc run() = {
    var b: bool;
    b <$ duniform [false; true];
    log ← b :: log;
    return b;
  }
}.
```

Note that `BitSampler` does not initialize its `log` variable. In this case, the contents of this variable will depend on the initial memory. In EasyCrypt, the whole memory (state) of a program is referred to by `&m` (or `&n` etc.). If `A` is a module then 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 example, `glob BitSampler` equals to `bool list`, and `(glob Bitsampler){m}` is the same as `log{m}` which is the value of `log` variable in memory `&m`.

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}`.

Module Types. In EasyCrypt, module types specify the types of the procedures in a module, but say nothing about the global variables of the module.

For example, `BitSampler` can be typed as `Runnable`:

```
module type Runnable = {
  proc run(): bool
}.
```

Probability Expressions. EasyCrypt has built-in `Pr`-constructs which are used to express the probabilities of events in program executions. The general form of `Pr`-expression is as follows: `Pr[program @ initial memory: event]`. For example, `Pr[r ← BitSampler.run() @m: P r]` denotes the probability that the return value `r` of procedure `run` of module `BitSampler` given the initial memory **m** satisfies the predicate `P`.

In EasyCrypt, the program in `Pr`-notation can only be a single procedure call. To simplify the presentation, we relax this restriction and allow us to write multiple statements. In the actual EasyCrypt code the same can be expressed by defining module wrappers with a procedure that contains those statements.

To give an example, we can prove that for any adversary `A`, the success probability of guessing the output of a `BitSampler` is exactly ½. In the following we reuse the `Runnable` module type to universally quantify over adversaries. In EasyCrypt, the notation `M <: T` indicates that the module `M` satisfies the module type `T`.

```
lemma example: ∀ (A <: Runnable{-BitSampler}) m,
  Pr[b1 ← BitSampler.run();
     b2 ← A.run() @m: b1 = b2] = 1/2.
```

It is important to understand that the module type `Runnable` also includes adversaries (i.e., modules) that read from and/or write to `BitSampler`'s log (e.g., `BitSampler` itself). To exclude such "cheating" adversaries, EasyCrypt allows us to write `Runnable{-BitSampler}` to denote the subset of adversaries whose global variables are disjoint from those of `BitSampler`.

2.2. Rewinding

Rewinding refers to the proof technique in which we take a given (usually unknown) adversary (in EasyCrypt modelled as an abstract module) `A`, and convert it into an adversary `B` that in some form includes the following 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.

In [6], the authors explain that while the above steps seem simple there are numerous challenges in trying to express them in EasyCrypt.

The authors provide a solution to rewindability in EasyCrypt in the form of a generic library. In a nutshell, the authors argue that a module `A` is rewindable iff:

1) There exists an injective mapping `f` from $\mathcal{G}_A$ to some parameter type `sbits`. Intuitively, `sbits` is the type of bitstrings.
2) The module `A` must have a terminating side-effect free 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)`.

3) The module `A` must have a terminating procedure `setState`, so that whenever it is invoked with argument `x: sbits`, so that `x = f g` for some `g:` $\mathcal{G}_A$ then `A` must be set into a state `g`.

To express the above conditions formally, the authors define a module type `Rew` for rewindable modules:

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

(Here, the symbol $\star$ indicates that the procedure (re)initializes all global variables of a module.)

In our presentation, we use `Rewindable A` as a shorthand which indicates that `A` satisfies the rewindability condition explained above. The fully formal EasyCrypt definition of rewindability can be found in [6].

2.3. Running Example: The Fiat-Shamir Protocol

For the clarity of presentation we instantiate our formal definitions using the Fiat-Shamir zero-knowledge protocol as a running example [7]. (Not to be confused with the well-known Fiat-Shamir transformation from the same paper.) The language of Fiat-Shamir protocol consists of quadratic residues. An element $s \in \mathbb{Z}/n\mathbb{Z}$ is a quadratic residue if there exists w so that $s = w^2$ and s is invertible.[2] In Fiat-Shamir protocol the prover tries to convince a verifier that a statement is quadratic residue and it knows the witness.

Let us give an informal protocol description. The protocol starts by the prover generating a random invertible ring element r and sending its square $a = r^2$ to the verifier. The verifier receives the commitment a and replies with a random bit b as a challenge. The prover replies with $z = w^b r$. Finally, the verifier accepts if $z^2 = s^b a$ and a is invertible.

In the following sections we use this protocol as a concrete running example for which we derive completeness, special soundness, soundness, proof of knowledge, zero-knowledge, and sequential compositionality.

In Sec. 6, we comment on our formalisation of other protocols.

3. Generic Definitions

In this section we formalize main definitions which are associated with sigma protocols. In cryptgoraphy, there are three types of definitions, namely, perfect, statistical, and computational. Let us describe these definitions in broader sense. In *perfect* definitions the adversarial party usually has unlimited computational capabilities and the probability of successful attack must be zero. In *statistical* definitions the adversarial party is still unlimited, but the probability of successful attack could be non-zero but small. In *computational* definitions the adversary is computationally limited and we also allow a non-zero probability of a successful attack. In this paper we mainly present statistical definitions, but in our formalization we also address computational and perfect variations.

2. In this section the multipication must be understood as a ring multiplication.

In our formalization we define an EasyCrypt theory which encompasses definitions of types, operators, modules, and modules types from this section. Later the EasyCrypt cloning mechanism can be used to instantiate these definitions for a specific protocol. The lemmas in this section must be understood as definitions of properties of sigma protocols (i.e., proof obligations for concrete instances).

3.1. Basics

From an abstract point of view, every sigma protocol is designed to work with a specific formal NP-language. The language is induced by a relation between statements and witnesses. More specifically, a language is a subset of statements for which there exists a witness which satisfies the relation.

```
type statement, witness.

type relation = statement → witness → bool.

op in_language (R:relation)(s:statement): bool
  = ∃ (w: witness), R s w.
```

Informally, in sigma protocols the prover tries to convince the verifier that it knows a witness which validates the statement (i.e., satisfies the relation of a language).

It is important to note that in some cases the proofs of properties of sigma protocols such as completeness, soundness, and zero-knowledge could require relations of different strength. Therefore, in our library when the user instantiates their protocol we ask them to provide the relation per property.

```
op completeness_relation: relation.
op soundness_relation: relation.
op zk_relation: relation.
```

In the following sections, we formally describe the main properties associated with sigma protocols. We start by only expressing these properties relative to a single run, i.e. three messages. To achieve reasonable security guarantees most sigma protocols are executed multiple times. Therefore, we show that the one run execution properties can be lifted to multiple runs generically (see Sec. 5).

3.1.1. Fiat-Shamir Basics. In our formalization we express Fiat-Shamir in terms of an abstract ring $\mathbb{Z}/n\mathbb{Z}$ whose elements have type `zmod`. The standard library of EasyCrypt features a theory `ZModRing` with an extensive formalization of properties of `zmod`.

To increase readability we will use type synonyms `qr_stat`, `qr_wit`, `qr_com`, and `qr_resp` for the statement, witness, and the response, respectively. All these types are synonyms of `zmod`.

The Fiat-Shamir language consists of statements which are quadratic residues in `zmod`. On the formal side we need to define relations for completeness, soundness, and zero-knowledge. All three relations are the same and they ensure that statement is a square of the witness and also that the statement is invertible:

```
op completeness_relation (s: qr_stat)(w: qr_wit)
  = s = w * w ∧ invertible s.

op zk_relation        = completeness_relation.
op soundness_relation = completeness_relation.
```

3.2. Completeness

The sigma protocol consists of a honest prover and a honest verifier. In our library, we give generic module types for both parties:

```
module type HonestProver = {
  proc commitment(s: statement,
                  w: witness): commitment
  proc response(ch: challenge): response
}.

module type HonestVerifier = {
  proc challenge(s: statement,
                 c: commitment): challenge
  proc verify(r: response): bool
}.
```

In terms of sigma protocols, the `commitment` procedure produces the first message, `challenge` produces the second message, and `response` the third. Finally, given the response the verify procedure decides whether the verifier accepts.

We implement the following module `Completeness` that encodes exactly this exchange of messages and returns whether the verifier accepts:

```
module Completeness(P: HonestProver,
                    V: HonestVerifier) = {
  proc run(s: statement, w: witness): bool = {
    var c, ch, r, acc;
    c   <@ P.commitment(s,w);
    ch  <@ V.challenge(s,c);
    r   <@ P.response(ch);
    acc <@ V.verify(r);
    return acc;
  }
}.
```

It is important to understand that the sigma protocol is *defined* by the implementation of honest prover (which we denote by `HP`) and the honest verifier (which we denote by `HV`).

The honest verifier of a sigma protocol must choose its challenge uniformly at random from some finite set. Also, the verification procedure can usually be defined as a predicate (pure function) on the statements and transcripts (the transcript is a triple of commitment, challenge, and response). Therefore, we give a "skeleton" implementation of a honest verifier which can be instantiated by providing protocol specific `challenge_set` and `verify_transcript` operators:

```
type transcript = commitment × challenge × response.

module HV: HonestVerifier = {
  var s, c, ch;
  proc challenge(s: statement, c: commitment) = {
    (HV.s, HV.c) ← (s,c);    % global state vars
    ch <$ duniform challenge_set;
    return ch;
  }
  proc verify(r: response): bool = {
    return verify_transcript s (c, ch, r);
  }
}.
```

Intuitively, the sigma protocol induced by the honest prover `HP` and honest verifier `HV` is complete iff for any valid statement `s` the probability of success in `Completeness(HP,HV)` game is close to one. The `completeness_error` is a protocol specific error term which determines the probability of failure of `Completeness`.

```
lemma completeness:
  ∀ (s: statement)(w: witness) &m,
    completeness_relation s w ⇒
      Pr[r ← Completeness(HP,HV).run(s,w) @m: r]
        ≥ 1 - completeness_error s.
```

(This "lemma" must be understood as a definition of completeness as a property.)

3.2.1. Fiat-Shamir Completeness. In this section, we formally define the Fiat-Shamir protocol by implementing an honest prover and an honest verifier.

We start by implementing the honest prover. In the commitment phase the prover samples an invertible group element `r` uniformly at random and returns its square as the commitment. The value `r` and the witness `w` are stored in the prover's internal variables `HP.r` and `HP.w`, respectively:

```
module HP: HonestProver = {
  var r, w: zmod
  proc commitment(s: qr_stat,
        w: qr_wit): qr_com = {
    HP.w ← w;
    r <$ zmod_distr;
    return r · r;
  }
  proc response(b: bool): qr_resp = {
    return (if b then r · w else r);
  }
}.
```

To instantiate the implementation of the honest verifier we need to define the set of challenges and the verification function. In the Fiat-Shamir protocol the verifier's challenge is just a bit, hence, the challenge set consists of values `false` and `true`:

```
op challenge_set = [false; true].
```

The verification function starts by checking that the statement and the prover's commitment are invertible and then checks that in case when challenge bit is `false` the square of the response value equals to the commitment (`c`) value, otherwise the square of the response must equal to the product of the commitment and the statement.

```
op verify_transcript (s:qr_stat)(t:transcript)
 = let (c, ch, r) = (t.1, t.2, t.3) in
     invertible s ∧ invertible c
      ∧ (if ch then c · s else c) = r · r.
```

At this stage the Fiat-Shamir protocol is fully defined by the implemented honest prover `HP` and instantiated honest verifier `HV`.

In our formalization we prove that the protocol has "perfect" completeness, i.e., the completeness error is zero. With the help of SMT solvers, EasyCrypt is able to derive completeness almost entirely automatically.

3.3. Soundness

Soundness is an important property of sigma protocols which says that if the statement is false (not in the language of the sigma protocol) then cheating prover cannot convince an honest verifier that it is true, except with some small probability.

The module type `MaliciousProver` defines the interface of cheating provers. Note that the main difference from `HonestProver` is that the `commitment` procedure of the cheating prover only receives the statement (since in the context of the soundness property the witness for the provided statement does not exist).

```
module type MaliciousProver = {
  proc commitment(s: statement): commitment
  proc response(ch: challenge): response
}.
```

Similarly to the module `Completeness` we implement a module `Soundness` which encodes one run of a sigma protocol in the context of a cheating prover.

```
module Soundness(P: MaliciousProver,
                 V: HonestVerifier) = {
  proc run(s: statement): bool = {
    var c, ch, r, acc;
    c   <@ P.commitment(s);
    ch  <@ V.challenge(s,c);
    r   <@ P.response(ch);
    acc <@ V.verify(r);
    return acc;
  }
}.
```

The sigma protocol is *statistically sound* iff for any cheating prover `P` and a statement `s` which is not in the language induced by `soundness_relation` the probability that the honest verifier accepts in the `Soundness` game is bounded from above by some small `soundness_error s`. Here, `soundness_error` is a protocol specific function that is allowed to depend on the statement `s`.

```
lemma soundness:
  ∀ (s: statement) (P <: MaliciousProver) m,
    !(in_language soundness_relation s) ⇒
      Pr[r ← Soundness(P,HV).run(s) @m: r]
        ≤ soundness_error s.
```

(This "lemma" must be understood as a definition of statistical soundness as a property.) The *perfect soundness* would be similar to the statistical soundness with `soundness_error s` being defined as zero. However, we are not aware of any interesting sigma protocols which achieve perfect soundness.

In the case of *computational soundness*, the soundness-error depends on the computational power of the malicious prover `P`. That is, the right-hand-side becomes a protocol-specific term that depends on the success of `P` in a different game (the "reduction").

3.3.1. Fiat-Shamir Soundness.

The Fiat-Shamir protocol is statistically sound with soundness-error equal to ½. In our formalization we derive this from extractability by using the generially derived lemma (see Sec. 4.3 and Sec. 4.3.1).

3.4. Special Soundness

For some sigma protocols the easiest way to prove soundness is to derive it from another property known as "special soundness".

The main idea of special soundness is that if for the same statement we have two valid transcripts for the same commitment but with different challenges, then it should be possible to efficiently extract the witness from these transcripts. Recall that the transcript is a triple of commitment, challenge, and response.

The function `valid_transcript_pair s t₁ t₂` checks whether transcripts satisfy the condition stated above:

```
op valid_transcript_pair
  (s: statement) (t₁ t₂: transcript): bool
   = t₁.1 = t₂.1
     ∧ t₁.2 ≠ t₂.2
     ∧ verify_transcript s t₁
     ∧ verify_transcript s t₂.

op special_soundness_extract
  (s: statement) (t₁ t₂: transcript): witness.
```

The function `special_soundness_extract` is protocol specific and must be instantiated by the user.

The most intuitive variant is the *perfect special soundness*. It states that the function `special_soundness_extract` must be able to construct a valid witness from any valid transcript pair.

```
lemma perfect_special_soundness:
  ∀ (s: statement) (t₁ t₂: transcript),
    valid_transcript_pair s t₁ t₂ ⇒
      soundness_relation s
        (special_soundness_extract s t₁ t₂).
```

For the computational case, we also additionally need to define a module type for special soundness adversaries:

```
module type SpecialSoundnessAdversary = {
  proc attack(s: statement):
    transcript × transcript
}.
```

Intuitively, *computational special soundness* states that for any computationally limited adversary it must be hard to derive a pair of valid transcripts for which the `special_soundness_extract` function fails to provide a valid witness. In EasyCrypt, we express this as an event whose probability is bounded from above by a small number `special_soundness_error A s`, where `A` is a special soundness adversary and `s` is a statement.

```
lemma computational_special_soundness: ∀ s m,
  Pr[r ← A.attack(s) @m:
    valid_transcript_pair s r.1 r.2
     ∧ !(soundness_relation s
          (special_soundness_extract s r))]
     ≤ special_soundness_error A s.
```

Unfortunately, in EasyCrypt one cannot define operators to depend on the modules (such as `special_soundness_error` above). As a result of this restriction the user must manually replace `soundness_error A s` with the error term in the above lemma.

We do not define statistical special soundness because it would be equivalent to perfect special soundness.[3]

3. If we do not have perfect special soundness, then there exists a valid transcript pair on which the deterministic extraction algorithm does not extract successfully. Therefore there exists a (possibly unbounded) algorithm that searches for such a transcript pair and outputs it. This algorithm succeeds with probability 1, so the scheme does not have statistical special soundness (for any soundness error < 1).

3.4.1. Fiat-Shamir Special Soundness. The Fiat-Shamir protocol has perfect special soundness. Let us define the extraction function:

```
op special_soundness_extract
  (s:qr_stat) (t₁ t₂:transcript): qr_wit =
    let (c₁,ch₁,r₁) = t₁ in
    let (c₂,ch₂,r₂) = t₂ in
      if ch₁ then r₁ · (inv r₂) else (inv r₁) · r₂.
```

The main idea is as follows. Let `t₁ := (c₁,ch₁,r₁)` and `t₂ := (c₂,ch₂,r₂)` be a pair of valid transcripts with respect to the function `valid_transcript_pair` (i.e., $c_1 = c_2$, $ch_1 \neq ch_2$, and both t_1 and t_2 pass the honest verification). Also, w.l.o.g. assume that $ch_1 = \texttt{true}$ and $ch_2 = \texttt{false}$. In this case we know that $c_1 \cdot s = r_1 \cdot r_1$ and $c_1 = r_2 \cdot r_2$ because the transcripts pass the verification. Therefore, $s = (r_1 \cdot (\texttt{inv}\ r_2)) \cdot (r_1 \cdot (\texttt{inv}\ r_2))$, i.e., the statement is a square and a witness is $r_1 \cdot (\texttt{inv}\ r_2)$.

For the given definition of `special_soundness_extract`, the perfect special soudness is derived almost entirely automatically by using the built-in support for SMT solvers.

3.5. Proof of Knowledge

Proof of knowledge, also known as *extractable* proof systems, guarantee that there exists an extractor which can compute a witness from a rewindable malicious prover. The extractor is parameterized by the prover and has access to its rewinding interface (see Sec. 2.2).

```
module type Extractor(P: RewMaliciousProver) = {
  proc extract(s: statement): witness
}.
```

Here, `RewMaliciousProver` is a module type for rewindable cheating provers which must implement both `MaliciousProver` and the rewindability interface.

The success of the extractor depends on the probability with which the prover manages to convince the honest verifier in the `Soundness(P,HV)` game.

In the general case, statistical extractability assumes that there exists an efficient `Extractor` so that:

```
lemma extractability:
  ∀ s m (P <: RewMaliciousProver),
    Rewindable P ⇒
      let sound_prob =
          Pr[r ← Soundness(P,HV).run(s) @m: r] in
        Pr[r ← Extractor(P).extract(s) @m:
            soundness_relation s r]
          ≥ extraction_success sound_prob s.
```

We assume that prover is rewindable (i.e., the `Rewindable P` premise, see Sec. 2.2 for details).

The function `extraction_success` specifies a lower bound on the success probability of the `Extractor`. `Extractor` and `extraction_success` are protocol-specific and must be provided by a user.

In the case of *computational extractability*, the extraction success depends on the computational power of the malicious prover `P`. That is, the right-hand-side becomes a protocol-specific term that depends on the success of `P` in some different game (the "reduction").

3.5.1. Fiat-Shamir Proof of Knowledge. The Fiat-Shamir protocol is a statistical proof of knowledge and in our formalization we derive this from the special soundness by using the generic lemma (see Sec. 4.2 and Sec. 4.2.1).

3.6. Zero-Knowledge

Zero-knowledge is a property of sigma protocols which ensures security guarantees for honest provers. This is achieved by expressing that malicious verifiers cannot get any "new information" about the witness of a statement from the communication with the honest prover that they would not be able to compute by themselves without that communication.

We model this by requiring that anything the malicious verifier learns (w.l.o.g., what it outputs) can be simulated by a "simulator" that knows everything the verifier knows, except for the witness. The simulation is successful if no "distinguisher" can tell the verifier's and the simulator's outputs apart (even when the distinguisher knows the witness).

Formally, the above is expressed by using two different games and a distingusher. The first game, `ZKReal`, implements the interaction between the honest prover and a malicious verifier. The main difference between a malicious verifier and an honest verifier is that after receiving the response from the prover, a malicious verifier computes a "summary"[4] of the entire interaction instead of outputting a success bit. Next, that summary is sent to the distinguisher together with the witness. The distinguisher outputs a bit which indicates whether the distinguisher thinks it was given a summary produced by the first or the second game (see below).

The second game, `ZKIdeal`, is parameterized by a simulator, a malicious verifier, and a distinguisher. In this game, the simulator is trying to produce a summary which the distinguisher would not be able to tell apart from the `ZKReal` case. It is important to note that the simulator must produce its summary without seeing the witness while the distinguisher gets the simulator's summary and the witness of the statement, same as in the `ZKReal` game. The simulator can internally run and rewind the malicious verifier. It does not interact with the prover. The simulator is protocol-specific and must be specified as part of the security proof.

In EasyCrypt, the aforementioned games are defined as follows:

```
module ZKReal(P: HonestProver,
              V: RewMaliciousVerifier,
              D: Distinguisher) = {
  proc run(s: statement, w: witness) = {
    var c, ch, r, sum, guess;
    c   <@ P.commitment(s, w);
    ch  <@ V.challenge(s, c);
    r   <@ P.response(ch);
    sum <@ V.summitup(r);
    guess <@ D.guess(s, w, sum);
    return guess;
  }
}.

module ZKIdeal(S: Simulator,
```

4. Our development is parameterized with a datatype `summary` which is supposed to hold the information about the protocol-run produced by the verifier.

```
                V: RewMaliciousVerifier,
                D: Distinguisher) = {
  proc run(s: statement, w: witness) = {
    var sum, guess;
    sum <@ S(V).simulate(s);
    guess <@ D.guess(s, w, sum);
    return guess;
  }
}.
```

The sigma protocol has *statistical zero-knowledge* iff there exists an efficient simulator `Sim` such that for any statement `s` witnessed by `w`, any rewindable malicious verifier `V`, and any distinguisher `D`, the absolute difference between success probabilities of `ZKReal(HP,V,D)` and `ZKIdeal(Sim,V,D)` is bounded from above by `zk_function s`. Here, `zk_function` is a protocol specific and depends on the statement `s`.

```
lemma zero_knowledge:
  ∀ s w n m (V <: RewMaliciousVerifier)
      (D <: Distinguisher),
  zk_relation s w ⇒
    |Pr[r ← ZKReal(HP,V,D).run(s,w)@m: r]
        - Pr[r ← ZKIdeal(Sim,V,D).run(s,w)@m: r]|
      ≤ zk_function s.
```

In case of *perfect zero-knowledge* the success probabilities of `ZKReal` and `ZKIdeal` must be equal,[5] and in case of *computational zero-knowledge* the right-hand-side of the inequality can additionally depend on `V`.

In the case of *computational zero-knowledge*, the right-hand-side of the inequality depends on the computational power of the malicious verifier `V` and distinguisher `D`.

It is also possible to have another variant of statistical zero-knowledge where the verifier is computationally bounded, but the distinguisher is computationally unbounded. This is encoded exactly like computational ZK, except that the right hand side of the inequality depends only on the malicious verifier `V` but not the distinguisher `D`.

3.6.1. Fiat-Shamir Zero-Knowledge. The Fiat-Shamir protocol has statistical zero-knowledge and in our formalization we derive this by using the “one-shot” simulators and our generic lemmas (see Sec. 4.1 and Sec. 4.1.1).

4. Generic Derivations

In the previous section we introduced security properties associated with sigma protocols. For some protocols it can be challenging to prove properties like soundness, zero-knowledge, and extractability directly. Therefore, one often derives these properties from simpler ones using generic derivations. We formalize three of the most important such derivations. More specifically, in Sec. 4.1 we derive zero-knowledge from the existance of a “one-shot” simulator, in Sec. 4.2 we derive extractability from special soundness, and in Sec. 4.3 we derive soundness from extractability.

4.1. Zero-Knowledge from One-Shot Simulation

In Sec. 3.6, we introduced the zero-knowledge property in which a distinguisher compares the protocol-summary generated by the malicious verifier to the summary produced by a simulator.

In practice, to prove zero-knowledge, one usually starts by defining a “one-shot simulator” which produces a simulated summary but may abort with some relatively high probability (e.g., ½). Conditioned on not aborting (the “success”-event) that simulator’s output must be indistinguishable from the real protocol interaction. Later, the actual zero-knowledge simulator runs and rewinds the one-shot simulator until the “success”-event happens. In this section, we generically address this transformation. In the end, a user must only implement a “one-shot simulator”, prove its indistinguishability conditioned on a “success”-event, and establish a lower bound of the “success”-event. Then the zero-knowledge proprety of its iterated version is implied automatically by our lemmas.

A one-shot simulator is a module parameterized by a rewindable malicious verifier. It has a `run` procedure which takes the statement and returns a pair of a boolean and a protocol-summary. The boolean indicates whether the “success”-event mentioned above happened:

```
module type Simulator1(V:RewMaliciousVerifier) = {
  proc run(s: statement): bool × summary
}.
```

For the rest of this section, we fix a rewindable malicious verifier `V`, a distinguisher `D`, and a one-shot simulator `Sim1`. Our derivation works for any `V`, `D`, and `Sim1`. Also, `Sim1` will typically depend on the protocol and will be specified explicitly by the user. Depending on the variant of ZK we are analyzing (i.e., perfect, statistical or computational), we then consider over unlimited or computationally bounded `V`, `Sim1`, and `D`.

For the sake of readability, we introduce an abbreviation `sim1_dist_prob` which denotes the probability that both the “success”-event happens and the distinguisher outputs `true`:

```
abbrev sim1_dist_prob(s, w, m): real =
 Pr[(success, sum) ← Sim1(V).run(s);
   guess ← D.guess(s, w, sum) @m: success ∧ guess].
```

The main property associated with `Sim1` is that the probability `sim1_dist_prob(s,w,m)` conditioned on the “success”-event of `Sim1` is at most ϵ away from the probability that the distinguisher outputs `true` in the `ZKReal(HP,V,D)` game. Here, ϵ is a protocol specific real number. The conditional probability is expressed as a ratio.

```
op ϵ : real.

axiom sim1_dist_prob_prop: ∀ s w m,
  zk_relation s w ⇒
    |Pr[r ← ZKReal(HP, V, D).run(s, w) @m: r]
      - (sim1_cond_prob(s, w, m)
        / Pr[(success, _) ← Sim1(V).run(s) @m:
          success])| ≤ ϵ.
```

This “axiom” must be understood as a property of `Sim1` which must be proved by the user.

Now we generically implement a simulator `SimN` which wraps the one-shot simulator and runs it until the “success”-event occurs, but at most `N` times, where `N` is a

5. In the literature, we also find a different weaker definition of perfect zero-knowledge (e.g., [20, Definition 4.3.1]). This definition in fact simply states the existence of a one-shot simulator with *zero* distinguishing probability. So this definition is also covered by our work using the definitions from Sec. 3.6.

parameter.[6]

```
module SimN(Sim1: Simulator1)
    (V: RewMaliciousVerifier): Simulator = {
  proc run(s:statement,w:witness) = {
    var c ← 0;
    var success ← false;
    var summary;
    while (c < N ∧ !success){
      (summary, success) <@ Sim1(V).run(s);
      c ← c + 1;
    }
    return (summary,success);
  }
}
```

Note that Sim1 in SimN.run may modify its state when executed. This means that in the second iteration of the loop Sim1 might run on an invalid initial state (and no guarantees can be made). To avoid this we would need to rewind Sim1. To support this, the user would need to prove the technical condition Rewindable Sim1 (see Sec. 2.2). While possible in principle, this approach would lead to additional boilerplate. Instead we found it more convenient to simply request user to ensure the following property which guarantees that Sim1 itself rewinds its state when it is not successful[7]:

```
axiom sim1_rew: ∀ m s,
  Pr[(succ, _) ← Sim1(V).run(s) @m:
    !succ ⇒ G^fin_Sim1(V) = G^m_Sim1(V)] = 1.
```

The third and final property associated with Sim1 is existance of σ which is a lower bound on the "success"-event:

```
op σ : real.

axiom succ_event_prob: ∀ m s,
  Pr[(succ, _) ← Sim1(V).run(s) @m: succ] ≥ σ.
```

The main result of this section states that SimN(Sim1) is a simulator whose success probability in the ZKIdeal game is $\epsilon + 2(1-\sigma)^N$-close to the success probability of V in the ZKReal game, where σ is the lower bound on the probability of the "success"-event of Sim1, and N is a number of iterations performed by SimN:

```
lemma statistical_zk: ∀ s w m,
  zk_relation s w ⇒
    |Pr[r ← ZKReal(HP, V, D).run(s, w) @m: r]
       - Pr[r ← ZKIdeal(SimN(Sim1),
           V, D).run(s, w) @m: r]|
      ≤ ϵ + 2 · (1 - σ)^N.
```

4.1.1. Fiat-Shamir Zero-Knowledge. In this section, we show how to derive zero-knowledge from a one-shot simulator for the Fiat-Shamir protocol. The main idea behind one-shot simulators is to "guess" the challenge of the verifier and then prepare a "special" commitment such that the simulator is able to correctly respond to the guessed challenge (and only that challenge) even without knowing the witness. The simulator aborts when it guessed incorrectly (i.e., "success"-event is the correct guess of the simulator).

In the Fiat-Shamir the challenge is a bit, so the one-shot simulator tries to guess the challenge by uniformly sampling a bit b. If it sampled false, the one-time simulator outputs r · r as the commitment, where r is a uniformly sampled invertible element. If the sampled bit b is true, the one-time simulator additionally multiplies r · r with the inverse of the statement (i.e., r · r · (inv s)). For both challenges, the corresponding response r is valid from the honest verifier's point of view. Thus, r is sent to the verifer. However, if the challenge given by verifier was not the same as the bit guessed by the simulator then the simulator rewinds the verifier back to its initial state.

```
module Sim1(V: RewMaliciousVerifier) = {
  proc run(s: qr_prob): bool × adv_summary = {
    var r, z, b', b, result, vstate, r, rr, bb;
    r <$ zmod_dist;
    b <$ duniform [false; true];
    c ← if b then r · r · (inv s) else r · r;
    vstate <@ V.getState();
    b' <@ V.challenge(s, c);
    result <@ V.summitup(r);
    if (b ≠ b') {
      V.setState(vstate);
    }
    return (b = b', result);
  }
}.
```

Now, to conclude the proof of the zero-knowledge property for Fiat-Shamir protocol, we are only left with three proof obligations. The first one is that in case when the "success"-event does not happen (i.e., b ≠ b'), the state of Sim1(V) does not change. For Sim1 this is easily shown by using the probabilistic Hoare logic and the assumption that V is rewindable.

The second proof obligation (i.e., succ_event_prob property) is to find σ which is a lower bound on the "success"-event (i.e., b = b'). Observe that in Sim1 the values b and b' are not independent since the commitment c depends on b and b' is computed based on c. However, in the proof we can lose this dependency by observing that the values r · r and r · r · (inv s) are distributed equally. As a result, we can show that "success"-event occurs with probability exactly equal to ½.

For the third proof obligation we need to define ϵ and prove the lemma sim1_dist_prob_prop described in Sec. 4.1 which is enough to conclude statistical zero-knowledge for Fiat-Shamir by application of our statistical_zk result. It turns out that for Fiat-Shamir protocol the ϵ is zero and we can derive the following formula:

```
|Pr[r ← ZKReal(HP, V, D).run(s, w) @m: r]
  - sim1_cond_prob(s, w, m)
    / Pr[(succ, _) ← Sim1(V).run(s) @m: succ]| = 0.
```

The main observation here is that conditioned on the "success"-event the protocol summaries in one-time simulator and in the ZKReal are distributed equally.

4.2. Extractability from Special Soundness

The goal of this section is for protocols with special soundness to implement a generic knowledge-extraction

6. It would also be possible not to enforce an upper bound. Then simulator would have finite expected runtime, but no *a priori* bound on the worst-case runtime.

7. This does not remove the need for rewindability because to prove sim1_rew, Sim1 will need to rewind V.

module which is parameterized by a rewindable malicious prover `P` and then relate the lower bound of the extractor's success with the success probability of `P` in the `Soundness(P,HV)` game.

Intuitively, the goal is to show that if a malicious prover `P` is "too successful" in winning the `Soundness(P,HV)` game, then it knows the witness. More precisely, there is a generic extractor that will be able to compute a witness from `P` with sufficiently high probability (assuming that `P` is rewindable) after the success probability in the `Soundness(P,HV)` reaches a "cut-off" point, called the "knowledge error".

In [6], the authors use EasyCrypt to derive the security of a coin-toss protocol from the following generic lemma:

```
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.
```

The lemma states that the probability of success (according to some predicate `M`) in two sequential runs of `A.run` is lower-bounded by the square of the probability of success in a single run. Note that this even holds in the presence of an initialization `B.init` that is called once. The presence of `B.init` is what makes the lemma technically non-trivial.

This property is also important for sigma-protocols: One can instantiate `B.init`, `A.run`, and the predicate `M` so that the "initialize-then-single-run" case will exactly correspond to the `Soundness(P,HV)` game. More specifically, `B.init` must be instatiated as the prover's comitment phase and `A.run` as the remaining message-exchanges between `P` and the honest verifer (i.e., challenge, response, and verify). With that in mind if we examine the success event of the left-hand-side of the `rew_with_init` inequality, we will see that its success event corresponds to two transcripts passing the verification. In cases when transcripts have distinct challenges these transcripts are "valid" from the perspective of the special soundness extractor (see Sec. 3.4) and we can attempt to extract a witness using the `special_soundness_extract` function. Recall that honest verifier samples challenges uniformly at random and therefore the probability of having distinct challenges at the transcripts is always high (exponential in the bit-length of the challenge). Based on the description above, we implement a generic extractor parameterized by a rewindable malicious prover:

```
module Extractor(P: RewMaliciousProver) = {
  proc extract(s: statement): witness = {
    var i, c1, c2, r1, r2, pstate;
    i <@ P.commitment(s);
    pstate <@ P.getState();
    c1 <$ duniform challenge_set;
    r1 <@ P.response(c1);
    P.setState(pstate);
    c2 <$ duniform challenge_set;
    r2 <@ P.response(c2);

    return special_soundness_extract s
      (i, c1, r1) (i, c2, r2);
  }
}.
```

What remains is to analyze the probability of successful extraction by `Extractor(P)`. In our EasyCrypt formalization we show that the lower bound for the success probability of `Extractor(P)` depends on the size of the challenge set and the success probability of `P` in the soundness game. We do derivations for both computational and perfect special soundness. For the simplicity of presentation, we only present the latter result here:

```
lemma statistical_extractability: ∀ m s,
(∀(t1 t2: transcript),
    valid_transcript_pair s t1 t2
     ⇒ soundness_relation s
        (special_soundness_extract s t1 t2))
 ⇒ Pr[r ← Extractor(P).extract(s) @m:
      soundness_relation s r]
     ≥ (Pr[r ← Soundness(P, HV).run(s) @m: r]^2
      - 1 / (size challenge_set)
        · Pr[r ← Soundness(P, HV).run(s) @m: r]).
```

4.2.1. Fiat-Shamir Proof of Knowledge.

In Sec. 3.4.1 we explained that the Fiat-Shamir protocol has perfect special soundness. Therefore, we get the lower bound on extractability of the protocol automatically by applying the `statistical_extractability` lemma. More specifically, since the challenge for Fiat-Shamir is a boolean then for any malicious prover `P`, the statement `s` which is not in the language of soundness relation, and the initial state **m** we have:

```
Pr[r ← Extractor(P).extract(s) @m:
    soundness_relation s r]
  ≥ Pr[r ← Soundness(P, HV).run(s) @m: r]^2
    - 1/2 · Pr[r ← Soundness(P, HV).run(s) @m: r].
```

Note that this is larger than zero whenever the success probability in the soundness game is larger than ½. So the knowledge-error is ½.

4.3. Soundness from Extractability

In the previous section we explained how to generically derive extractability from special soundness. The probability of a successful witness extraction by `Extractor` module is lower-bounded by a function of the success probability of the malicious prover in the `Soundness` game. However, for statements which are not in the language, the witness extraction probability is zero by definition. These observations can be used to generically derive an upper bound for the soundness of a sigma protocol from its extractability.

To state our theorem we first need to specify the relationship between success probabilities of extractor and soundness games. We do so by fixing a function `f` such that there exists an ϵ, so that for any $f\ x \leq 0$, the value `x` is less than or equal to ϵ. This function depends on the bounds obtained when proving extractability and must be specified by the user. For example, if we derive extractability via perfect special soundness (see Sec. 4.2) `f` would be $\lambda\ x.\ x^2 - x/(\texttt{size challenge\_set})$. The main result of this section is the following lemma:

```
lemma statistical_soundness: ∀ m s f ϵ,
  ! in_language soundness_relation s ⇒
  let sound_prob
     = Pr[r ← Soundness(P,HV).run(s)@m: r] in
  ⇒ Pr[r ← Extractor(P).extract(s)@m:
```

```
                    soundness_relation s r]
        ≥ f sound_prob
  ⇒ (∀ (x : real), f x ≤ 0 ⇒ x ≤ ϵ)
  ⇒ sound_prob ≤ ϵ.
```

The upper bound on the soundness of sigma-protocols with perfect special soundness is a simple corollary from `statistical_soundness` and `statistical_extractability`:

```
lemma soundness_from_special_soundness: ∀ m s,
 (∀(t1 t2: transcript),
   valid_transcript_pair s t1 t2
    ⇒ soundness_relation s
       (special_soundness_extract s t1 t2))
  ⇒ ! in_language soundness_relation s
  ⇒ Pr[r ← Soundness(P,HV).run(s)@m: r]
      ≤ 1/size challenge_set.
```

4.3.1. Fiat-Shamir Soundness. In Sec. 4.2.1 we automatically derived extractability from special soundness. Similarly, we now can apply `soundness_from_special_soundness` and get an upper bound on the soundness-error of Fiat-Shamir. More specifically, for any malicious prover `P` and statement `s` (not in the language of `soundness_relation`) the soundness-error is below ½:

```
Pr[r ← Soundness(P, HV).run(s) @m: r] ≤ 1/2.
```

5. Sequential Composition

In previous sections we introduced properties associated with one run of sigma-protocols. In practice one run of the sigma protocol usually does not provide sufficient security guarantees. For example, in our running example (Fiat-Shamir protocol), the soundness-error could be bounded from above by ½ which means that even in case when statement is not in the language, a cheating prover can succeed half of the time.

To solve this, a standard approach is sequential repetition of the protocol. If we have a sigma protocol (P, V) with soundness-error δ, then the probability that the prover succeeds n times in a row is δ^n. So, given a sigma protocol, we get a better proof system (P^n, V^n) by repeating it n times.[8] But then, we need to ask the question: Does (P^n, V^n) still have completeness? Still zero-knowledge?

The answer is fortunately yes (completeness and zero-knowledge degrade linearly), and in this section we prove this.

Since (P^n, V^n) is not a sigma protocol (it exchanges more than three messages), we cannot directly apply the definitions of completeness, soundness, and zero-knowledge from the previous section to it. (There is no problem in principle, it is just that the games are specifically formulated for protocols with three messages.) One solution would be to generalize the definitions so that they can apply to protocols that send an arbitrary number of messages. At a first glance, this seems trivial, but encoding such protocols is slightly awkward: All messages would need to have the same type, and we have to somehow encode when the protocol stops, and we might have to ask what happens when one participant stops before the other does, etc. To avoid this, we choose a slightly simpler approach: Instead of trying to define (P^n, V^n) generically and apply generic definitions to it, we directly hardcode the sequential repetition into our definitions. For example, iterated completeness would be a definition that is parametrized by (P, V), and that runs (P, V) exactly n times and then checks whether all runs were successful. This leads to somewhat less general definitions but makes the presentation more consise, and is sufficient for our use-case of sequential repetition of sigma protocols.

In the following we address security bounds of sequential composition for completeness, soundness, and zero-knowledge. We leave proof of knowledge for the future work as it is more complicated to approach formally.

5.1. Iterated Completeness

We start by defining a module `CompletenessAmp` which iterates the `Completeness` module `n` times, where `n` is a parameter. The resulting bit indicates whether or not all runs were successful.

```
module CompletenessAmp(P: HonestProver,
                       V: HonestVerifier) = {
  proc run(s: statement, w: witness, n: int) = {
    var accept, i;
    i ← 0;
    accept ← true;
    while (i < n ∧ accept) {
      accept <@ Completeness(P, V).run(s, w);
      i ← i + 1;
    }
    return accept;
  }
}.
```

The iterated completeness states that if success probability of one run of `Completeness` is bounded from below by δ, then `n` runs are bounded from below by δ^{n}:

```
lemma completeness_seq: ∀ m s w δ n,
  completeness_relation s w
 ⇒ 1 ≤ n
 ⇒ (∀ n,
      Pr[r ← Completeness(P,V).run(s,w)@n: r]
         ≥ δ)
 ⇒ Pr[r ← CompletenessAmp(P,V).run(s,w,n)@m: r]
         ≥ δ^n.
```

This result indicates that the success probability of having `n` successful runs degrades exponentially quickly. This suggests that sigma protocols will have "reasonable" levels of iterated completeness only if the one-run bound (i.e. δ) is close to one. Note that this does not mean that completeness-error grows exponentially quickly. Indeed, if the compleness-error is ϵ (i.e., δ=1−ϵ), then the completness-error for iterated case is $1-\delta^{n} \leq n \cdot \epsilon$, so the error grows only linearly.

5.1.1. Fiat-Shamir Iterated Completeness. Previously we explained that for Fiat-Shamir the completeness-error is zero. Therefore, as an immediate consequence of `completeness_seq` result we get that the completeness-error of iterated Fiat-Shamir is zero as well.

5.2. Iterated Soundness

In this section we argue that if we iterate the `Soundness` game then the probability of not "catch-

8. We consider only sequential repetition. Parallel repetition is considerably more complex and out of the scope of this work.

ing" a cheating prover on a non-succesful run decreases exponentially.

Similar to the iterated completeness we first define the `SoundnessAmp` game which iterates the `Soundness` module `n` times.

```
module SoundnessAmp(P: MaliciousProver,
                    V: HonestVerifier) = {
  proc run(s: statement, n: int) = {
    var accept, i;
    i ← 0;
    accept ← true;
    while (i < n ∧ accept) {
      accept <@ Soundness(P,V).run(s);
      i ← i + 1;
    }
    return accept;
  }
}.
```

It is important to note that the state of cheating prover could be different during every iteration because EasyCrypt allows procedures to keep state between activations. Thus the malicious prover is one program sending $2n$ messages, not an n-fold repetition of the same program. In contrast, the honest verifier does the same thing in each iteration by definition (see definition of `HV` in Sec. 3.2).

The statement of iterated soundness states that if the success probability of one run of the soundness game by cheating prover `P` and honest verifier `HV` (the "soundness-error") is bounded from above by δ, then iterating it `n` times improves this upper bound to $\delta^{\mathtt{n}}$:

```
lemma soundness_seq: ∀ m s δ n,
 ! in_language soundness_relation s
 ⇒ 1 ≤ n
 ⇒ (∀ n, Pr[r ← Soundness(P,HV).run(s)@n: r] ≤ δ)
 ⇒ Pr[r ← SoundnessAmp(P,HV).run(s,n)@m: r] ≤ δ^n.
```

5.2.1. Fiat-Shamir Iterated Soundness. Recall that we proved that for one-run case the soundness-error of Fiat-Shamir is upper-bounded by $\frac{1}{2}$. Hence, as an immediate consequence of `soundness_seq`, the upper bound for the soundness-error of iterated Fiat-Shamir is exponentially better, namely, $(\frac{1}{2})^n$.

5.3. Iterated Zero-Knowledge

Similarly to the case of completeness and soundness the goal of iterated zero-knowledge is to show that if the distinguishing probability for one round is small, it is also small in the case of multiple runs. In other words, we need to show that zero-knowledge composes sequentially.

We start by introducing a module `ZKRealAmp` which iterates one run of the "real" protocol `n` times. Note that instead of iterating the `ZKReal` module which has the distinguisher at the end, we first iterate the protocol `n` times and only then run the distinguisher who gets the summary prepared by the verifier after all `n` iterations (w.l.o.g., the return values of `V.summitup` in prior rounds are ignored). The main idea is that the verifier tries to accumulate as much information throughout the runs and then present that summary to a distinguisher:

```
module ZKRealAmp(P: HonestProver,
                 V: MaliciousVerifier,
                 D: Distinguisher) = {
  proc run(s: statement, w: witness) = {
    var c, ch, r, summary, guess, i;
    i ← 0;
    while (i < n) {
      c  <@ P.commitment(s, w);
      ch <@ V.challenge(s, c);
      r  <@ P.response(ch);
      summary <@ V.summitup(r);
      i ← i + 1;
    }
    guess <@ D.guess(s, w, summary);
    return guess;
  }
}.
```

In the ideal setting, both in the one run and in the iterated case, the game simply consists of a simulator that outputs the final output of the verifier; no interaction is happening. Thus, we can reuse the module `ZKIdeal` for the iterated case.

However, the concrete simulator `Sim` from the one run case will not properly simulate the multi-run case. Thus, we need to construct a new simulator `SimAmp(Sim)` for the iterated case from `Sim`. The following module `SimAmp` encodes this transformation:

```
module SimAmp(S: Simulator,
              V: MaliciousVerifier) = {
  proc simulate(s: statement) = {
    var summary, i;
    i ← 0;
    while (i < n) {
      summary <@ S(V).simulate(s);
      i ← i + 1;
    }
    return summary;
  }
}.
```

Note that the security definition of zero-knowledge does not require us to use this specific `SimAmp`, as long as we construct a simulator that simulates well. However, it is the most natural way of constructing the simulator of the multi-run case; it just repeats the simulator of the single-run case.[9]

We are now ready to introduce our main ZK iteration result. Let `Sim`, `V` and `D` be a simulator, malicious verifier, and a distinguisher, respectively. Let `Di(D)` denote a distinguisher which executes `S(V).simulate(s)` exactly `i` times and then calls `D.guess` and returns its result. (Here `i` is a global variable that belongs to `Di`.[10] For brevity, we omit the formal definition of `Di` here.)

If there exists a δ which is an upper bound for the distinguishing probability with respect to `Sim`, `V`, `D` and honest prover `HP`, then the difference between `ZKIdeal` experiment played by `SimAmp(Sim)` and the real amplified game `ZKRealAmp` played by `P`, `V`, and `D` is upper-bounded by $n\delta$. The most important aspect of this result is that the security degrades linearly with the number of performed runs of the protocol.

```
lemma zk_seq: ∀ m δ,
```

9. This construction makes use of the fact that the verifier's state is part of the simulator's state. The iterated simulator needs to "feed" the state from the previous iteration to the internally simulated verifier. However, in the present setting, this is automatic because the iterated simulator just keeps the state between iterations of the loop. This is why we do not see any explicit state passing between the iterations of the loop in `SimAmp`.

10. The variable `i` does not appear explicitly in the code below. This is because the arbitrary initial value of `i` is implicitly present in the memory **n**.

```
(∀ n, |Pr[r ← ZKIdeal(Sim,V,Di(D)).run(s,w)@n: r]
    - Pr[r ← ZKReal(HP,V,Di(D)).run(s,w)@n: r]|
        ≤ δ)
⇒ |Pr[r ← ZKIdeal(SimAmp(Sim),V,D).run(s,w)@m: r]
   - Pr[r ← ZKRealAmp(HP,V,D).run(s,w)@m: r]|
     ≤ δ · n.
```

The proof of this result is not as simple as the amplification of soundness and completeness. In fact, to get a linear bound, we used a proof based on hybrid argument. It is fortunate that the standard library of EasyCrypt contains a generic formalization of this technique.

5.3.1. Running Example: Iterated Zero-Knowledge. Recall that for single-run case we showed that the zero-knowledge-error of Fiat-Shamir is upper-bounded by $2(1-p)^N$, where N is the number of iterations performed by the simulator. Hence, as an immediate consequence of zk_seq lemma, the upper bound for the zero-knowledge-error of iterated Fiat-Shamir is only linearly worse, namely, $2n(1-p)^N$, where n is the number of sequential runs of Fiat-Shamir.

6. Case Studies

Throughout this paper we instantiated our definitions and lemmas with the Fiat-Shamir protocol (Sec. 2.3). We proved directly completeness (Sec. 3.2.1), special soundness (Sec. 3.4), and one-shot simulator property (Sec. 4.1.1). Due to the algebraic nature of the Fiat-Shamir protocol the built-in support for SMT solvers greatly simplified these proofs. Most importantly, (iterated) completenss, (iterated) soundness, (iterated) zero-knowledge, and proof of knowledge were implied automatically by our generic results.

In this section we additionally comment on the Schnorr and Blum protocols.

6.1. Schnorr Protocol

The protocol is defined for a cyclic group G_q of order q with generator g. The language of the Schnorr protocol consists of all group elements. In the Schnorr protocol the prover tries to convince a verifier that it knows a discrete logarithm of the statement. In other words, if $s \in G_q$ is a statement then then a corresponding witness is an element w so that $s = g^w$. The group G_q and the generator g are public parameters.

The prover interacts with the verifier as follows:

1) The prover chooses a $y \in \mathbb{Z}_q$ uniformly at random and sends $z := g^y$ as the commitment.
2) The verifier replies with a challenge $c \in \mathbb{Z}_q$ chosen uniformly at random.
3) The prover responds with $t = y + cw$.
4) The verifier accepts if $g^t = zs^c$.

Completeness. The Schnorr protocol has perfect completeness. For our EasyCrypt implementation the completeness proof is derived almost entirely automatically by using the built-in support for SMT solvers.

Soundness. It is important to note that in Schnorr protocol all statements (i.e., elements of the group) are in the language (i.e., have witnesses). Therefore the protocol is trivially sound with soundness-error 0. Since soundness is thus a meaningless property for this protocol, we do not prove it. But note that it is meaningful to say that the protocol proves that the prover *knows* a witness. We show this below (proof of knowledge).

Zero-Knowledge. Another interesting aspect of the Schnorr protocol is that we cannot show the malicious-verifier zero-knowledge property because there is no efficient simulator.[11] Indeed, the challenge of the verifier is an element of the group sampled uniformly at random. This means that the size of the set of challenges (challenge_set) equals to the order of the group (i.e., exponentially big). Hence, if we build a simulator based on the idea of guessing the challenge of a malicious verifier then the probability of a successful simulation will be negligible.

The Schnorr protocol does, however, satisfy the honest-verifier zero-knowledge (HVZK) property. This property has been formalized in [11], [13], [14] and is not related to rewinding, so we did not include a proof in our case study.

Special Soundness. The Schnorr protocol has perfect special soundness. Recall that in the case of perfect special soundness, we are given two transcripts $t_1 = (c_1, ch_1, r_1)$ and $t_2 = (c_1, ch_2, r_2)$ which have the same commitments $(c_1 = c_2)$ and different challenges $(ch_1 \neq ch_2)$. In addition we know that both transcripts pass the verification which tells us that $r_1 = y + c_1 w$ and $r_2 = y + c_2 w$. Hence, witness can be extracted by computing $(r_1 - r_2)(c_1 - c_2)^{-1}$. In our formalization we used this idea to implement the extraction function and the special soundness is derived almost entirely automatically by the built-in SMT solvers.

Proof of Knowledge. From special soundness we conclude the proof of knowledge property automatically by using the previously derived lemma statistical_extractability. We get the following lower bound on the success probability of the extractor:

```
Pr[r ← Extractor(P).extract(s)@m: s = g^r]
  ≥ (Pr[r ← Soundness(P,HV).run(s)@m: r]^2
    - 1/(size challenge_set)
        · Pr[r ← Soundness(P,HV).run(s)@m: r])
```

It is important to note that since challenge_set $= \mathbb{Z}_p$ is big then 1/(size challenge_set) is small and therefore the success probability of an extractor is close to the square of the success probability of malicious prover in the soundness game.

The Schnorr protocol illustrates that the size of the challenge set affects zero-knowledge and proof of knowledge differently. The small challenge set is good for zero-knowledge property since the simulator has more odds of guessing the verifier's challenge; on the other hand, the small challenge set makes extractor less efficient (and vice versa for the big challenge set).

6.2. Blum Protocol

A graph G is said to be Hamiltonian if there exists a cycle that passes through each vertex of G exactly once. This cycle is called a Hamiltonian cycle. Finding a Hamiltonian cycle in a graph is an NP-complete problem. Blum described a zero-knowledge protocol whose language

11. At least no construction is known. We are not aware of a formal impossibility result.

consists of Hamiltonian graphs. More specifically, the statements are graphs and witnesses are Hamiltonian cycles.

In the following we describe the Blum protocol. Let the graph G have n vertices and be represented by the adjacency matrix with n rows and n columns. We write $G(i, j)$ to refer to the entry at the i-th row and j-th column. The entries of adjacency matrix are booleans so that $G(i, j) = 1$ iff there is an edge from vertex i to vertex j. We also assume *(Commit, Verify)* is a commitment scheme. In the Blum protocol the prover P wishes to prove to the verifier V that G is Hamiltonian as follows:

1) The prover samples a random permutation ϕ over n vertices. Then it uses ϕ to construct $\phi(G)$ which is a permuted version of the adjacency matrix of G. Next, P commits to each edge in the permuted adjacency matrix. Let $C_\phi(i, j)$ denote a commitment resulting from running the *Commit* algorithm on the $\phi(G)(i, j)$, so C_ϕ is a matrix of commitments. The opening keys of these commitments are stored in a matrix D_ϕ. Additionally, P commits to the permutation ϕ that was used to permute the adjacency matrix; denote this commitment by p_ϕ and its respective opening key by q_ϕ. Finally, the prover proceeds by sending p_ϕ and C_ϕ to the verifier.
2) The verifer replies with a challenge bit $b \in \{0, 1\}$ sampled uniformly at random.
3) The prover receives the bit b. If $b = 0$ then prover sends the permutation ϕ and opening keys D_ϕ and q_ϕ to the verifier. That is, it opens commitments to the permuted graph and to the permutation. Otherwise, if $b = 1$ then prover uses the permutation ϕ on w which translates it to the Hamiltonian cycle in $\phi(G)$. Finally, the prover sends $\phi(w)$ and the opening keys from D_ϕ for only those entries whose edges appear in $\phi(w)$.
4) If the challenge bit was zero (i.e., $b = 0$) then verifier receives a permutation ϕ, opening keys D_ϕ and q_ϕ. It uses the commitment verification algorithm *Verify* to check that q_ϕ is a correct opening for ϕ and that D_ϕ is a correct opening for $\phi(G)$.
   However, if $b = 1$, then verifier receives a cycle $\phi(w)$ and openings for its edges. It checks that the received cycle is Hamiltonian and that the openings corresponding to the edges of $\phi(G)$ are all 1. (That is, that $\phi(G)$ is indeed a subgraph of the committed graph.)

Commitment Scheme. The security properties of the Blum protocol depend on the hiding and binding properties of the underlying commitment scheme. More specifically, special soundness, extractability, and soundness depend on the binding property, and zero-knowledge depends on the hiding property of the commitment scheme. In our formalization we assumed that the commitment scheme is statistically hiding and computationally binding.[12] As a result we proved that our variant of Blum protocol has computational special soundness, extractability, and soundness. At the same time zero-knowledge is statistical.

12. We choose this direction because in the context of this paper, computational soundness is more interesting that computational ZK: Analyzing computational soundness showcases the use of rewinding in the computational setting (i.e., in a setting involving reductions).

Completeness. The completeness of the Blum protocol relies on the correct operation of the commitment scheme. In our formalization we assumed that the commitment scheme always produces functional commitment-opening pairs. In this case the Blum protocol has perfect completeness.

Special Soundness. The special soundness of the Blum protocol depends on the binding property of the used commitment scheme. Let $t_1 = (c_1, ch_1, r_1)$ and $t_2 = (c_2, ch_2, r_2)$ be two valid transcripts (i.e., $c_1 = c_2$ and $ch_1 \neq ch_2$). Then w.l.o.g., assume that $ch_1 = 0$ and $ch_2 = 1$. In this case, $c_1 = (C_\phi, q_\phi)$, where C_ϕ contains commitments on edges for a permuted graph, and q_ϕ is a commitment on the permutation. Also, $r_1 = (\phi, D_\phi, q_\phi)$, where ϕ is a permutation, D_ϕ contains valid opening keys to commitments in C_ϕ and q_ϕ is a valid opening to p_ϕ. At the same time, $r_2 = (w', d)$ where w' is a Hamiltonian cycle in the permuted graph and d contains openings of the w'-edges in C_ϕ. At this point, we known that the keys in d successfully open commitments in C_ϕ and that w' is a cycle. We also know that keys in D_ϕ open commitments in C_ϕ with respect to $\phi(G)$. Hence, either $\phi^{-1}(w')$ is a Hamiltonian cycle in G or there exists a commitment in C which corresponds to an edge in w' which opens to two different values by the respective keys from d and D.

Based on this idea we implemented a function `special_soundness_extract` which returns $\phi^{-1}(w')$ as a Hamiltonian cycle for G. Also, we define a `BlumBinder` module which finds two opening keys for different values and the same commitment in case when $\phi^{-1}(w')$ is not a Hamiltonian cycle for G.

To sum up, the probability of unsuccesfull witness extraction from two valid transcripts is bounded from above by the probability of a successfull attack on the binding of a commitment scheme:

```
lemma blum_spec_soundness: ∀ m,
 Pr[Extractor(P).extract(s)@m:
      valid_transcript_pair s r.1 r.2 ∧
       !soundness_relation s
        (special_soundness_extract s r.1 r.2) ]
 ≤ Pr[r ← BindingExperiment(
          BlumBinder(Extractor(P))).main()@m: r].
```

In our formalization, the **proof of knowledge** and **soundness** of Blum protocol are automatically derived by our generic lemmas from `blum_spec_soundness`.

Zero-Knowledge. In our formalization we derived zero-knowledge from a "one-shot" simulator. Similarly to the Fiat-Shamir protocol, in the Blum protocol we define a "one-shot" simulator which starts by trying to guess the challenge of the verifier. Next, assuming that the guess is correct, the simulator prepares a commitment to which it is going to be able to provide a response which will verify correctly. More specifically, if the "one-shot" simulator guesses that the verifier sends a 0-challenge, then the simulator produces a commitment which consists of C_ϕ and p_ϕ computed exactly as described in the protocol. However, if the simulator guesses that the challenge is 1, then it sends C'_ϕ and p_ϕ to the verifier where C'_ϕ is a commitment matrix produced for the complete graph with n vertices. If the guess was correct and the verifier challenge is 1, the simulator can reply with opening keys which correspond to edges in Hamiltonian cycle $\phi(h_n)$, where h_n is a trivial Hamiltonian cycle in the complete graph H_n.

The proof of the properties for the described "one-shot" simulator are not trivial. In our formalization, we give a sequence of cryptographic games which establish that if malicious verifier acts differently in the simulation and the real interaction with the honest prover then this verifier can be reduced to an attacker which breaks the hiding property of the commitment scheme.

7. Formalization Caveats

Initialization. One important design decision that arose in the formalization was how the initial state of the various adversarial algorithms (e.g., malicious verifier, malicious prover) is instantiated. We identified several options:

(a) Adversarial algorithms get a special procedure called `init()` whose task is to initialize their state.
(b) Adversarial algorithms get a special procedure called `init()`, and additionally this procedure is constrained to initialize all variables of the adversary before reading them. (No dependence on the initial memory **m**. EasyCrypt's module type system allows us to enforce this by adding a $\star$ to the procedure declaration.)
(c) Adversarial algorithms get no initialization procedure.
(d) Any of the three options above, and the adversary additionally get an all-quantified auxiliary input as an argument.

Different choices have different subtle consequences both on the cryptographic interpretation as well as on the details of the formal proofs.

Cryptographically, giving an additional all-quantified argument (known as auxiliary input) gives us a definition called *non-uniform zero-knowledge*. In contrast, without auxiliary input, we get *uniform zero-knowledge*. It is known that to get sequential composition of zero-knowledge proofs, we need to use non-uniform zero-knowledge [21]. So which of the above design options lead to a non-uniform definition? Obviously, option (d) has an auxiliary input. However, options (a) and (c) also have one, this is just implicit in the way how EasyCrypt works: Unless we explicitly enforce procedures that do not look at their initial state (as in (b)), all procedures can access the content of their global variables in the initial memory **m**. And all our theorems are of the form "$\forall$ **m**, ...", which means that the adversary effectively gets an auxiliary input implicitly. Only option (b) (when not combined with (d)) models uniform zero-knowledge. We believe that it is important to stress this point explicitly because EasyCrypt's handling of global variables makes it easy to overlook this implicit dependency.

In the formal setting it is easier to work with games without explicit state initialization. For example, in our development we defined the `Soundness(P,HV)` game which encodes the three message exchange between the malicious prover `P` and the honest verifier `HV`. Later, we defined the module `SoundnessAmp` which sequentially iterates the `Soundness` game n times. Then we proved that n-time sequential composition of sigma-protocols exponentially reduces the soundness-error to δ^n. The proof is based on the premise that for any initial state the soundness-error for the `Soundness` game is below δ. However, if we add state initialization of to the malicious prover in the `Soundness` game, then it is meaningless to keep `SoundnessAmp` defined as an n-time iteration of the `Soundness` game. Instead, we will need to add an explicit initialization of the malicious prover before the while-loop and the body of the while-loop must implement the three message exchange. This means that the proof of amplified soundness (similar to lemma `soundness_seq`) will become more complicated due to the fact that we split the prover into an iterated and a non-iterated part (e.g., we will need to use "averaging" technique and Jensen's inequality).

In our formalization we used the (c) approach as it results in simpler proofs. However, we recognize that in some situations the state initialization of adversaries is necessary. To make our results relevant for these situations, we provide generic lemmas for removing/adding init-procedures in security claims (not limited to the zero-knowledge setting).[13]

Disjointness of Module-Variables. In the cryptographic setting, when we have a definition such as zero-knowledge, and we say "there is a simulator S such that for all verifiers V and all distinguishers D, ...", we usually implicitly mean that those three algorithms S, V, D have disjoint state. That is, we expect that they do not access each other's variables unless we would explicitly specify that they do so. When, e.g., the simulator runs a simulated V internally, we then can think of that V as a *copy* of the original V inside the simulator. The simulator still has no access to the variables of the "original" V. In the case of the simulator and the verifier, this distinction is of lesser importance because in the games making up the definition of zero-knowledge, S and V never run in the same game, so they cannot influence each other anyway. However, the distinguisher and the verifier, for example, run at the same time. So it can make a difference whether they can read/write each other's variables or not. The implicit assumption in the cryptographic setting is again that they do not. And if the distinguisher D needs, e.g., to simulate internally a copy of the verifier V (e.g., in the sequential composition proof), it is understood that this is a completely separate instance of V whose variables are part of the state of D.

Translating this to the EasyCrypt setting, the obvious solution would be to restrict the global variables of the various algorithms analogously. That is, we quantify over `S{-V,-D}`, `D{-S,-V}`, `V{-S,-D}`, which EasyCrypt understands to mean that their global variables are disjoint. Doing so, however, we encountered a problem that has no counterpart in the cryptographic pen-and-paper proofs: In the sequential composition proof for zero-knowledge, we needed to construct a distinguisher D that internally simulates a copy of V. However, EasyCrypt has no support for "copying" a module. Instead, if D wants to depend on the behavior of V, D has to access the module V directly,

13. We proved a generic lemma which states that for any algorithm `A` if there exists an ϵ which is an upper/lower bound for the probability of the "initialize-then-single-run" program then there exists a memory **n** (initial state) so that running `A` on **n** (without explicit initialization) results in the success-probability also bounded from above/below by ϵ. We also prove an analogous one for indistinguishability.

and this access can modify the state of V.[14] In order to be able to prove sequential composition, we thus needed to relax the condition on the variables of D and V, and allow the distinguisher and verifier to have common global variables. (I.e., declaring `D{-S}`, `V{-S}`.)

Since this departs from what the cryptographer expects, it is important to check whether this changes the meaning of the zero-knowledge definition. Fortunately, in the specific case of zero-knowledge, it does not, because the distinguisher runs after the verifier has already terminated, so their code does not "get into each other's way". Also, the goal of the verifier is to output as much information as possible about what it learned, so we can assume without loss of generality that the verifier tells everything to the distinguisher anyway. So the fact that the distinguisher can read the verifier's state does not give any information to the distinguisher that it should not have.

So in the present situation, all is fine if we relax the disjointness conditions. However, in other contexts (maybe some proof where a distinguishing entity runs concurrently with an adversary such as in the Universal Composability framework [22]), it might not be possibly to allow different modules to share state for technical reasons. We believe that it would be a very useful feature if EasyCrypt would allow us to "copy" modules to facilitate proofs where one program simulates another.

8. Conclusions

In this paper we focused on the formalization of sigma-protocols in EasyCrypt with a particular focus on rewinding. First, we formalized definitions of the main properties associated with sigma-protocols. Second, we used a formalization of the rewinding technique to generically derive malicious-verifier zero-knowledge, soundness, and proof of knowledge from simpler properties. Third, we addressed the sequential composition of sigma-protocols and derived security bounds for completeness, soundness, and zero-knowledge. Also, we used the Fiat-Shamir protocol as a running example and showed its instantiation in our framework. We also commented on our formalization of the Schnorr and Blum protocols. To the best of our knowledge, the properties of sigma-protocols which rely on rewinding of adversaries have not yet been addressed in theorem provers.

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.

14. We can, if we want, undo this modification using the `setState` and `getState` procedures (see Sec. 2.2). While this will make sure that semantically D does not change the state of V, it does not change which variables EasyCrypt *thinks* that D accesses because this is computed syntactically.

[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. Firsov and D. Unruh, "Reflection, rewinding, and coin-toss in easycrypt," in *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2022, pp. 166–179.

[7] A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems," in *Conference on the theory and application of cryptographic techniques*. Springer, 1986, pp. 186–194.

[8] C.-P. Schnorr, "Efficient signature generation by smart cards," *Journal of cryptology*, vol. 4, no. 3, pp. 161–174, 1991.

[9] M. Blum, "How to prove a theorem so no one else can claim it," in *Proceedings of the International Congress of Mathematicians*, vol. 1. Citeseer, 1986, p. 2.

[10] "Accompanying EasyCrypt development," https://github.com/dfirsov/easycrypt-zk-code, accessed: 2022-06-7.

[11] G. Barthe, D. Hedin, S. Z. Béguelin, B. Grégoire, and S. Heraud, "A machine-checked formalization of sigma-protocols," in *2010 23rd IEEE Computer Security Foundations Symposium*. IEEE, 2010, pp. 246–260.

[12] J. B. Almeida, E. Bangerter, M. Barbosa, S. Krenn, A.-R. Sadeghi, and T. Schneider, "A certifying compiler for zero-knowledge proofs of knowledge based on σ-protocols," in *European Symposium on Research in Computer Security*. Springer, 2010, pp. 151–167.

[13] J. Bacelar Almeida, M. Barbosa, E. Bangerter, G. Barthe, S. Krenn, and S. Zanella Béguelin, "Full proof cryptography: verifiable compilation of efficient zero-knowledge protocols," in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 488–500.

[14] D. Butler, A. Lochbihler, D. Aspinall, and A. Gascón, "Formalising $\sum$-protocols and commitment schemes using CryptHOL," *Journal of Automated Reasoning*, vol. 65, no. 4, pp. 521–567, 2021.

[15] J. B. Almeida, M. Barbosa, M. L. Correia, K. Eldefrawy, S. Graham-Lengrand, H. Pacheco, and V. Pereira, "Machine-checked ZKP for NP relations: Formally verified security proofs and implementations of MPC-in-the-head," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2587–2600.

[16] N. Sidorenco, S. Oechsner, and B. Spitters, "Formal security analysis of MPC-in-the-head zero-knowledge protocols," in *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. IEEE, 2021, pp. 1–14.

[17] I. Giacomelli, J. Madsen, and C. Orlandi, "ZKBoo: Faster Zero-Knowledge for boolean circuits," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 1069–1083.

[18] 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.

[19] ——, "EasyCrypt: A tutorial," in *Foundations of security analysis and design VII*. Springer, 2013, pp. 146–166.

[20] O. Goldreich, *Foundations of Cryptography, Volume 1*. Cambridge university press Cambridge, 2001.

[21] O. Goldreich and H. Krawczyk, "On the composition of zero-knowledge proof systems," *SIAM Journal on Computing*, vol. 25, no. 1, pp. 169–192, 1996.

[22] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. IEEE, 2001, pp. 136–145.