VCVio: A Formally Verified Forking Lemma and Fiat-Shamir Transform, via a Flexible and Expressive Oracle Representation

Devon Tuma, Nicholas Hopper
University of Minnesota

***Abstract*—As cryptographic protocols continue to become more complex and specialized, their security proofs have grown more complex as well, making manual verification of their correctness more difficult. Formal verification via proof assistants has become a popular approach to solving this, by allowing researchers to write security proofs that can be verified correct by a computer.**

In this paper we present a new framework of this kind for verifying security proofs, taking a foundational approach to representing and reasoning about protocols. We implement our framework in the Lean programming language, and give a number of security proofs to demonstrate that our system is both powerful and usable, with comparable automation to similar systems.

Our framework is especially focused on reasoning about and manipulating oracle access, and we demonstrate the usefulness of this approach by implementing both a general forking lemma and a version of the Fiat-Shamir transform for sigma protocols. As a simple case study we then instantiate these to an implementation of a Schnorr-like signature scheme.

1. Introduction

Designing secure cryptographic protocols is often error prone, and even small edge cases in proofs of their security can lead to major security vulnerabilities. Implementation errors are especially common, for example a recent paper by Dao et al. showed significant numbers of proof systems still using a weak and insecure version of the Fiat-Shamir transform. [1]. Computer-aided cryptography attempts to help mitigate these problems by providing computer-checked verification of a cryptographic protocol or implementation. This can range from verifying the mathematical correctness of a security reduction to verifying that a particular implementation is safe from timing attacks.

In this paper we focus on verifying the mathematical correctness of cryptographic security proofs. We specifically take a computational approach, where protocols are viewed as algorithms acting on bit-strings and adversaries are viewed as being (as powerful as) Turing machines. This is as opposed to a symbolic approach which reasons about protocols via a manually defined equational theory, which is meant to semantically capture the potential behavior without considering explicit algorithms. Most mechanized analyses of large-scale protocols have been done in the symbolic model using systems like Tamarin [?] or ProVerif [?], which make simplifying assumptions that aid in automation but provide weaker assurances about a protocol's security. While the computational approach often allows for less automation than the symbolic one, it provides more confidence in the correctness of the verification it produces, which is valuable given that limitations in equational theories have led to real world vulnerabilities being missed by symbolic verification. ([2], [3]).

A wide variety of frameworks have been developed to reason about cryptographic proofs in the computational setting, which all have different benefits and drawbacks depending on the specific use cases. These include EasyCrypt [4], CryptoVerif [5] SSProve [6], FCF [7], CertiCrypt [8], and CryptHOL [9]. Usually computations in this approach are represented by a shallow embedding into a more general proof assistant like Coq or Isabelle. Depending on the specific framework proofs can either be written as high-level asymptotic proofs about computational indistinguishability, or as low-level proofs of concrete probability bounds.

We do note that higher level and lower level approaches can be used together by implementing a more high-level framework as an abstraction over a more foundational one (For example EasyUC is built on top of EasyCrypt). This provides a way to reduce the amount of trust needed in the foundations of a higher level framework, as they are verified correct with respect to a much simpler foundation. For example the designers of the OWL system [10] have suggested this approach as a way to verify the currently axiomatic foundations of their high level system. SSProve similarly has a fully foundational base for its higher level abstractions. Our implementation provides a simple abstraction layer in order to reason about security games and adversaries, however in future work it would be useful to implement a more comprehensive high-level abstraction layer.

One important limitation with existing foundational systems is how they reason about the oracles available to a computation. In FCF for example oracle access is specified by a single oracle with fixed input and output types. This makes it difficult to reason about situations where multiple (or varying) oracles are available, e.g. both a signing oracle and a random oracle. One potential solution is the use of dependent sum types for the domain and range of the oracles, however this approach forces users to consider the possibility that a query could return an output for the 'wrong' oracle, requiring many failure checks. Nesting monads is also a potential solution, with one layer for each oracle, however this is cumbersome and can't handle infinite families of oracles (e.g. in defining unforgeability of ring signatures the adversary has a unique oracle for each ring size). In CryptHOL, oracles are implemented as "generative probabilistic values" and require restructuring adversaries and experiments into a series of smaller computations that run between oracle calls. Because adversaries cannot access the state of oracles, CryptHOL cannot easily express reductions in which oracles are reactively "programmed" or rewound to intermediate states by one adversary (such as an adversary that solves a hard computational problem) in response to the queries or results output by another adversary, such as an adversary that outputs a signature forgery.

In this work we define a framework with a generalized notion of oracle access that overcomes these existing limitations and implement this framework in the Lean programming language. Oracles in our system are parameterized by some (possibly infinite) indexing set, and the types of the input and output are given as a dependent function of the index. Computations themselves are represented as a sequence of purely syntactic oracle calls, with continuation functions specifying what to do with the result of the query. This structure is captured by a shallow embedding as a free monad type. [1]

This approach makes it feasible to treat probabilistic computation as a special case of having access to a coin-flipping oracle, and in particular this makes it possible to reason simultaneously about random behavior and other oracles. This unification can be very useful, which we demonstrate by using it to prove a general forking lemma, roughly based on the version given by Bellare and Neven [11]. Previous work by Firsov and Unruh [12] implemented a rewinding mechanism in EasyCrypt by introducing a system of probabilistic reflection from high level EasyCrypt to a more foundational representation, however their construction is somewhat limited by the nature of EasyCrypt itself. In particular they restrict to only a subclass of EasyCrypt modules and only allow an adversary to be rewound to it a fixed query. Our construction on the other hand allows us to fork an arbitrary adversary to arbitrary queries during its execution.

1. Other systems have used the term 'Free Monad' to refer to any monad where the bind and return operations are purely syntactic, without any further relations/semantics imposed upon them. A more general notion of the 'free monad of a functor' also exists, which has an important benefit of satisfying the basic monad laws (left identity, right identity, and associativity). Our approach lies somewhere between these two, and does satisfy these laws.

As a consequence of this we are able to give a verified Fiat-Shamir heuristic for sigma protocols, and to our knowledge signatures of this kind have not been verified in the computational setting by any previous work. Finally we instantiate this to get an implementation of a Schnorr-style signature algorithm.

On the other hand our generalization requires a more complicated semantics, which could lead to more complicated proofs and worse usability. It also exacerbates some existing limitations of foundational frameworks that in other approaches can mostly go unnoticed. We present a number of mechanisms to handle these issues, including the use of automatic type coercions and meta programming for proof generation. With these methods our system achieves comparable proof complexity to other foundational frameworks in practice, although it still offers less automation than many higher-level approaches. See here for our full implementation.

1.1. Contributions

To summarize, our main contributions are:

- Defining and implementing a new foundational framework for protocol verification with a focus on specifying and manipulating oracles.
- Solving several usability/automation issues that arise.
- Constructing a forking mechanism significantly more general than in any previous work.
- Formalizing a version of the Fiat-Shamir Heuristic for sigma protocols

1.2. Organization

We first give general background on the Lean programming language in section 2. Then in section 3 we define our representation of computations, and in section 4 and section 5 we give its denotational and operational semantics respectively. In section 6 we discuss adversaries, security games, and hardness assumptions.

Section 7 gives a construction of our forking lemma, and Section 8 gives our Fiat-Shamir construction. section 8 uses these two constructions to prove security of Schnorr Signatures. Finally in section 9 we discuss other verification frameworks and related work.

2. The Lean Proof Assistant

The Lean Theorem Prover is a dependently typed programming language and interactive proof assistant that enables computer verification of both mathematical proofs and properties of programs, similar to Coq or Isabelle. We implement our protocol in Lean and will use Lean syntax throughout the paper. We give a basic introduction to Lean here, but refer to the Lean Manual for a more comprehensive introduction.

At a basic level Lean is a functional programming language, and its syntax is similar to languages like Haskell or OCaml. The core logic of Lean is based on a type theory called the calculus of constructions (CIC), where every expression is a term and every term has a type, denoted `(a : A)`. The type of a function from type `A` to type `B` is written `A → B`, and function terms are expressed using lambda notation:

```
def foo : ℕ → ℕ → ℕ := λ x y ↦ x * y + 1
```

Functions can also be written by using `·` as a "hole" in an expression, for example the above could be written `(· * · + 1)`. Functions can be dependently type, so the type of an argument (or the output type) can depend on previous arguments to the function. For example the function `List.cons` is polymorphic in the type of the list's elements, which we express by naming the type of the first argument:

```
List.cons : {A : Type} → List A → A → List A
```

The curly braces indicate that when this function is called Lean should try to automatically determine the value of `A` based on type inference. This allows us to write `List.cons xs x` whether the elements of `xs` are strings, natural numbers, or something else.

The standard algebraic data types are built into the main language. Product types are written `A × B` with canonical elements `(a, b)` for `a : A` and `b : B`. Sum types are written `A ⊕ B` with canonical elements `inl a` and `inr b`. The singleton type `Unit` has a single canonical element `()`, and the empty type `⊥` has no elements. We also have a type `Bool`, with canonical elements `true` and `false`.

In order to be able to write and verify proofs, Lean also has a type `Prop` to represent mathematical propositions, and we view the propositions in `Prop` as themselves being types. Under this framework, an element `P : Prop` is a mathematical statement, and the "elements" of `P` are the proofs of that statement (This is essentially just the Curry-Howard isomorphism).

So for example given `(P Q : Prop)` the elements of `P ∧ Q` are pairs `⟨p, q⟩` of proofs `p : P` and `q : Q`. Similarly `P ∨ Q` has elements that are either a proof of `P` or of `Q`, and `P → Q` has elements that are functions from proofs of `P` to proofs of `Q`. We also have propositions `True` with a single element and `False` with no elements (note the capitalization compared to the elements of `Bool`). The negation `¬ P` is defined to be `P → False`.

While it's possible to write proofs as explicit terms in the language, lean also offers an interactive proof environment that allows proofs to be written via *tactic programming*. A tactic proof consists of a number of steps that modify the current "goal" to be proven, either solving it explicitly or generating new goal(s) that would suffice to complete the proof. The initial goal is the statement being proved, and a proof is complete if no goals remain. Many tactics are built in to the core language, but they can also be defined by users. These tactics can range in complexity from `contradiction` (which just starts a proof by contradiction), to `linarith` (which can check whether current linear constraints are satisfiable), and they can be sequenced together easily:

```
example (x y z : ℚ) (h1 : 2*x < 3*y)
    (h2 : -4*x + 2*z < 0) : 12*y - 4*z ≥ 0 :=
by contradiction; linarith
```

One particularly useful tactic is `simp`, which takes in a list of equalities or iff statements, and attempts to replace any instances of the left hand side of the equality in the goal with the right hand side. As an example we can use it to show that if $x+y = z$ for some non-negative y, then either $x < z$ or y is 0:

```
example (x y z : ℝ) (hy : 0 ≤ y)
      (hz : x + y = z) : x < z ∨ y = 0 :=
  by simp [← hz, le_iff_lt_or_eq, hy]
```

New lemmas can also be tagged to be automatically included in the list of lemmas used by `simp`, and building a good API of such lemmas can help significantly with proof automation. The main purpose of this tactic is to help reduce goals to a canonical form, however in many cases this canonical form ends up being `True` and the entire proof can be completed.

One other advantage of using Lean is the existing mathlib library [13], a large project that implements significant amounts of mathematical theory in a unified and interoperable way. We make use of this for representing concepts such as probability mass functions, group actions and computational complexity. Most newly written mathematical theory used in our system was written directly as contributions to mathlib.

3. Computations with Oracle Access

In order to reason about cryptographic protocols, we define a shallow embedding of computations with oracle access into the Lean type system, using a free monad to augment regular Lean functions with queries to oracles. This approach is most similar to that of FCF, and our monad is in some sense a unified generalization of the two monads used in that system.

3.1. Specifying Oracles

Before defining our model of computation we give a way to specify the set of oracles available to a computation. Specifications are given by a structure `OracleSpec ι`, where `ι` is an indexing set for the oracles. Each element `i : ι` corresponds to a unique oracle, and `domain i` and `range i` are the input and output types of the oracle corresponding to `i`:

```
structure OracleSpec (ι : Type) where
  domain : ι → Type
  range : ι → Type
```

We also require that all the involved types have decidable equality and that the output type of each oracle is non-empty, however these instances are generally handled automatically by the type-class system so we omit them here for simplicity. Note that we don't specify any behavior for these oracles, this should rather be seen as analogous to a type signature for the set of oracles that can be queried.

We define `singletonSpec T U` (denoted `T →ₒ U`) to represent access to a single oracle with input type `T` and output type `U`, by using the singleton type `Unit` for the indexing set and have the `domain` and `range` functions return `T` and `U` respectively:

```
def singletonSpec (T U : Type) :
    OracleSpec Unit where
  domain := λ () ↦ T
  range := λ () ↦ U
```

In order to represent probabilistic computation we define two additional oracle sets. Firstly `coinSpec` that gives access to a single coin flipping oracle returning a `Bool`, and secondly `unifSpec` that gives access to an infinite family of oracles, indexed over ℕ, where the nth oracle chooses a random value between 0 and n inclusively (represented by the type `Fin (n + 1)` in Lean). Since the input to these oracles is irrelevant, in both cases we use `Unit` for the domain types:

```
def coinSpec : OracleSpec Unit :=
  Unit →ₒ Bool

def unifSpec : OracleSpec ℕ where
  domain := λ n ↦ Unit
  range := λ n ↦ Fin (n + 1)
```

While it is possible to approximate uniform selection using only coin flips, it's generally simpler to use `unifSpec` and we will default to that going forward.

3.2. Representing Computations

We now define a language to represent computations with oracle access via an inductive type `OracleComp spec α`, where `spec` specifies what oracles the computation can make use of, and `α` gives the type of the final output. This type is a shallow embedding, so functions and expressions in this representation are just functions and expressions in the underlying language (Lean in our implementation), augmented with the ability to call oracles.

This embedding is done via a monad, which is a common way to represent computations with side effects in languages where all computations are pure. For example the `IO` monad gives a way to represent a computation that e.g. reads from `stdin` or writes to `stdout`. One way to think of of a value of type `IO α` is as a syntax tree for a computation with return type `α` that has access to syntactic read/write functions. In our case the "side effects" being captured by the monad are queries to the oracles.

We define this as an inductive type with only two constructors. The first is `pure' α x` which represents returning an pure Lean value `x : α`. The other, `queryBind' i t α oa`, represents querying oracle `i` on input `t` to get a result `u` and then running the computation `oa u`. Explicitly:

```
inductive OracleComp {ι : Type}
    (spec : OracleSpec ι) : Type → Type 1
| pure' (α : Type) (x : α) :
    OracleComp spec α
| queryBind' (i : ι) (t : spec.domain i)
    (α : Type) (oa : spec.range i →
      OracleComp spec α) : OracleComp spec α
```

We emphasize that the continuation `oa` in the `queryBind'` constructor is an arbitrary Lean function. Most of the "interesting" behavior of a computation is captured in this function as code in the underlying language, all that the monad structure adds to this is the ability to represent query calls.

We define `query i t` to be the computation that returns the result of querying oracle `i` on input `t`, directly returning the result as a pure value after:

```
def query (i : spec.ι) (t : spec.domain i) :
    OracleComp spec (spec.range i) :=
  queryBind' i t (spec.range i)
    (λ u ↦ pure' (spec.range i) x)
```

The computations `coin` and `\$[0..n]` are defined as special cases of this. Using these we can further define uniform selection from lists, finite sets, types, etc. For example uniform selection from a list is implemented by choosing a random index `k` and returning the `k`th element. We overload notation and write `\$ xs` for random selection from any type of collection.

We next define the general monadic bind operation on this type. The definition `bind' α β oa ob` will represent running the computation `oa` to get a result `x : α`, then running `ob` on input `x` to get a result of type `β`. We define this by induction on the first computation. If the first computation `oa` is a pure value, we insert it directly into the remaining computation. If the first computation is a query bound to a continuation, we move the second computation inside the continuation by a recursive call to `bind'`. Explicitly:

```
def bind' (α β : Type) : OracleComp spec α →
    (α → OracleComp spec β) →
    OracleComp spec β
  | pure' α a, ob => ob a
  | queryBind' i t α oa, ob =>
```

```
    query_bind' i t β (λ u ↦
      bind' α β (oa u) ob)
```

`OracleComp spec` with the operations `pure'` and `bind'` then forms a monad, and the three monad laws can all be verified easily by induction. We will generally use Lean's monad type class and its associated notation to write computations. In particular we can write `return a` for the pure operation, `oa >>= ob` for the bind operation, and use *do*-notation for sequencing larger computations:

```
example : OracleComp coinSpec ℝ := do
  let b ← coin; let b' ← coin
  let x := if b && b' then 3.141 else 0
  let y := if b || b' then 1.618 else 0
  return x * y
```

Additionally we have the monadic map operation `f <\$> oa` that runs the computation `oa` to get a result `x`, and returns `f x`. Similarly the sequence operation `og <*> oa` runs the computation `og` to get a *function* `g`, and then runs `g <\$> oa`. Both operations are syntactic sugar around the basic `return` and `>>=` operations. One very common use of this is to write `f <\$> oa <*> ob` for the computation that runs `oa` and `ob` separately to get `x` and `y`, returning only `f x y`.

We can also define computations recursively via pattern matching, assuming the recursion used is well-founded [2]. For example we can define a computation `replicate oa n` to repeat a computation n times, returning the result in a length n vector:

```
def replicate (oa : OracleComp spec α)
    (n : ℕ) : OracleComp spec (List α) :=
  match n with
  | 0 => return []
  | (n + 1) => (· :: ·) <$> oa
      <*> replicate oa n
```

3.3. Sub-Specs and Type Coercions

In order to combine sets of oracles we define an append operation on `OracleSpec`, to represent having access to oracles in either of the two original specs. We make use of sum types for the indexing set, with the `inl` and `inr` functions used to index the first and second sets of oracles respectively. The types of the domain and range at each index are defined by pattern matching on the provided index:

2. Like most proof verification frameworks Lean is limited to using well-founded recursion, ensuring that all recursive functions eventually terminate. Usually this amounts to allowing for structural recursion (e.g. recursing on the tail of a list) however Lean also allows custom annotations to prove termination. Generally cryptographic protocols avoid using unbounded recursion so this is rarely an issue, but we can still approximate such computations by taking a maximum "recursion-depth" argument as an input, and then considering the limit as this bound grows large, in which case the computation approaches the unbounded one.

```
def append (spec₁ spec₂ : OracleSpec) :
    OracleSpec (spec₁.ι ⊕ spec₂.ι) where
  domain := λ i ↦ match i with
    | inl i => spec₁.domain i
    | inr i => spec₂.domain i
  range := λ i ↦ match i with
    | inl i => spec₁.range i
    | inr i => spec₂.range i
```

We introduce the notation `spec ++ spec'` for this operation. We emphasize that this operation is neither commutative nor associative: while swapping the order doesn't change the set available oracles, it does change how you index into each of them.

One major issue in this representation is that it gives no way to sequence or combine computations where one has only a subset of the oracles of another. For example we can't bind `coin` to an adversary that has access to both a coin oracle and random oracle, since our monad definition requires that the `OracleSpec` remain fixed throughout a computation. Instead we are forced to make ad-hoc definitions of coin flipping for any new set of oracles. This presents major issues for modularity and code reuse, making the system harder to use in practice.

To solve this we create a system for automatically coercing the type of a computation to one with a larger set of oracles. We do this using Lean's `Coe A B` type-class, which specifies to Lean a way to automatically convert a value of type `A` to one of type `B` (e.g. `Coe ℚ ℝ` allows a rational number to be viewed as an element of the reals). Lean will automatically perform a type-class search for this whenever it fails to type check an expression, and apply a coercion if it finds one.

We implement these coercions for cases when the actual set of oracles is a *strict* subsequence of the expected set of oracles, allowing for arbitrary parenthesis in the sub-spec only. Note that we restrict to subsequences and not subsets of oracles, to avoid a potentially infinite type-class search that would come from allowing commutativity. The semantics we define in the next to sections are highly compatible with these coercions, and and our system is generally able to reduce a proof about a coerced computation down to an analogous proof about the original computation automatically, so we omit mention of them going forwards.

4. Probability Semantics

In this section we give a denotational semantics for `OracleComp`, where the denotation is a probability distribution modeling the probability of getting specific outputs from the computation. We only need to define this for computations on `unifSpec`, the behavior of other oracles can be reduced to this using the operational semantics we define later. However in some proofs it can be convenient to have denotations on any computation, to use as an intermediate step in a calculation. We therefore define it for arbitrary oracles by saying that *all* oracles available will respond uniformly at random (as they would for `unifSpec`).

Under this assumption we define a function `evalDist` from computations with return type `α` to probability mass functions on `α`, represented by the type `PMF α` (originally developed for use in the CryptoLib project [14]). `PMF` itself is a monad [3], and the mapping is a morphism of monads (i.e. it respects `pure` and `bind`). We define the `evalDist` of a query to be the uniform distribution on its output type.

We emphasizes that `evalDist` is neither injective nor surjective, as some distributions may have many possible implementations, while others may have none.

We will write `[= x | oa]` for the probability associated to `x` by `evalDist oa`, which gives a simple characterization of the semantics in practice:

```
[= x | return a] := if x = a then 1 else 0
[= y | oa >>= ob] :=
    Σ x, [= x | oa] * [= y | ob x]
[= u | query i t] := 1 / (spec.range i).card
```

The summation in the bind case given by `Σ` is a potentially infinite sum. This can lead to issues regarding convergence if probabilities are taken to be real numbers, so we instead use the type `ℝ≥0∞` of non-negative (potentially infinite) reals. All sums converge in this type (as they are increasing and bounded above by `∞`), and so this allows us to sidestep the issue entirely.

It's also possible to define a measure on `α` induced by a `PMF`, and we define the probability `[p | oa]` of some predicate `p` holding after running a computation as the measure of the (potentially infinite) set `{x | p x}`.

We define `support oa` to be the set of possible outputs of `oa`, i.e. the set of `x` such that `[= x | oa] ≠ 0`. We make heavy use of this when showing that an event has probability either `1` or `0`, allowing is to think about "possible outputs" rather than explicit probabilities. `Set` is also a monad [4], and it can be shown that this definition respects the monadic `pure` and `bind` operations. In some other systems this kind of reasoning is done with some form of relational Hoare logic, however we find that most proofs are already made simple enough by using `support`.

Finally we note that it's possible to define finite versions of the support using Lean's `Finset` type. This can be very useful in many proofs, however this does require that the output type of a computation has decidable equality (in order to delete duplicate elements of the set), and we notably don't have decidable equality for computations that return functions.

3. The pure operation is the indicator that has probability `1` at that point and `0` at all others. The bind operation is the distribution corresponding to drawing from the first of the two, and then drawing from the second with that output as input. Measures on (well-behaved) spaces also form a monad in a similar way

4. The pure operation is the singleton set, and the bind operation is an indexed union over the first set. This monad can be seen as representing non-deterministic computation, allowing a variable to be assigned an arbitrary number of values. The support of a computation can then be thought of as the result of assuming that oracle queries return all possible outputs at once.

5. Simulation Semantics

For oracles that aren't meant to respond uniformly, we now give an operational semantics which provides a method for simulating the behavior of oracles. The main construction is a function `simulate` that recursively substitutes the queries to an oracle with a specified implementation (that may have access to a different set of oracles). We allow the simulation to maintain some internal state, and augment the return value of the computation with the final state. For example, simulating a random oracle will consist of maintaining an internal cache of input/output pairs as the state, and the replacement will substitute queries to include a check to the cache before responding.

These semantics are generally used to reduce the set of oracles in a computation to a smaller set, but this isn't the only use case. For example logging an adversary's queries maintains the existing set of oracles, and our implementation of coercions is done by a simulation that grows the set of oracles.

5.1. Specifying Oracle Behavior

In order to represent a procedure for simulating a computation, we define a type `SimOracle spec spec' σ` for an implementation of the oracles in `spec` using a new set of oracles `spec'`, where `σ` is the type of an internal state shared throughout the simulation. This is given by a function that takes an oracle input and returns a new computation that should be used to replace the query:

```
def SimOracle (spec : OracleSpec)
    (spec' : OracleSpec) (σ : Type) :=
  (i : spec.ι) → spec.domain i → σ →
    OracleComp spec' (spec.range i × σ)
```

We introduce the notation `spec →[σ] spec'` for this type. We can then define a function `simulate` that applies a simulation oracle to a computation, recursively replacing queries with the new computations, passing the updated state along throughout, returning the final result of the computation and the final state value.

```
def simulate (so : spec →[σ] spec') :
    (oa : OracleComp spec α) → (s : σ) →
      OracleComp spec' (α × σ)
  -- Return the value and final state
  | pure' α x, s => return (x, s)
  -- Substitute the query and recurse
  | query_bind' i t α oa, s => do
      let (u, s') ← so i t s
      simulate so (oa u) s'
```

As a shorthand we will write `simulate'` for the version of `simulate` that discards the final state at the end. As a very simple example we have an identity oracle that substitutes back the original query, with a constant `Unit` element as the state:

```
def idOracle : spec →[Unit] spec :=
  λ i t () ↦ (·, ()) <$> query i t
```

This is an identity in the sense that we have exact equality between `simulate' idOracle oa ()` and the original computation `oa`. We have a very similar oracle for replacing oracle queries with a uniform choice of outputs, that can reduce an arbitrary set of oracles to `unifSpec`:

```
def unifOracle : spec →[Unit] unifSpec :=
  λ i t () ↦ (·, ()) <$> ($ spec.range i)
```

In this case we no longer have equality between the simulated and original computations (they are not even of the same type), but we do still have equality up to the previously defined probability semantics.

5.2. Combining Simulation Oracles

In order to represent more complex oracle implementations in a modular way, we define a number of ways to combine multiple simulation oracles. The first is a way to construct a combined simulation oracle for `spec₁ ++ spec₂` given individual simulation oracles for each. The resulting simulation oracle pattern matches the oracle index it receives and forwards to the corresponding simulation oracle, maintaining independent states for each of them:

```
def append (so : spec₁ →[σ] spec)
    (so' : spec₂ →[τ] spec) :
    spec₁ ++ spec₂ →[σ × τ] spec :=
  λ i ↦ match i with
    | (inl i) => λ t (s₁, s₂) ↦ do
        let (u, s₁') ← so i t s₁
        return (u, s₁', s₂)
    | (inr i) => λ t (s₁, s₂) ↦ do
        let (u, s₂') ← so' i t s₂
        return (u, s₁, s₂')
```

The other is a way to construct a simulation oracle that applies two simulation oracles in sequence, by simulating the simulation function of one with the other:

```
def compose (so : spec₁ →[σ] spec₂)
    (so' : spec₂ →[τ] spec) :
    spec₁ →[σ × τ] spec :=
  λ i t (s₁, s₂) ↦ do
      let ((t, s₁'), s₂') ←
        simulate so' (so i t s₁) s₂
      return (t, (s₁', s₂'))
```

We will write `++` and `∘` for these two operations. When combining many oracles with the above operations it's common to run into issues with extraneous state values, for example the state may have type `Unit × QueryLog` rather than the simpler (and isomorphic) type `QueryLog`. To avoid this we also define a function `maskState` that modifies the state type of a simulation oracle using a type equivalence (i.e. bijection) between some other state type. This amounts to just applying/removing the mask before/after running the simulation oracle (the function `Prod.map` applies two functions component-wise to a pair):

```
def maskState (so :  spec →[σ] spec)
    (e : σ ≃ τ) : spec →[τ] spec :=
  λ i t s ↦ map id e <$> so i t (e.symm s)
```

5.3. Counting, Logging, and Caching Queries

Besides just implementing oracle behavior, another important use case is tracking some information about a computation. For example we can easily implement a simulation oracles for logging the queries made by an oracle:

```
def loggingOracle : spec →[QueryLog] spec :=
  λ i t log ↦ do let u ← query i t
    return (u, log.logQuery i t u)
```

We also have a similar oracle for caching queries that logs fresh values, but returns old values if the input has been queried already:

```
def cachingOracle :
    spec →[QueryCache] spec :=
  λ i t cache ↦ match cache.lookup i t with
    | some u => return (u, cache)
    | none => do let u ← query i t
        return (u, cache.cacheQuery i t u)
```

Finally we can then define a random oracle applying this caching functionality, and then further reducing to a uniform selection oracle. This second reduction beyond caching introduces an extraneous `Unit` to the internal state which we mask away for simplicity:

```
def randOracle :
    spec →[QueryCache] unifSpec :=
  (unifOracle ∘ cachingOracle).maskState
    (Equiv.prodUnit (QueryCache spec))
```

6. Cryptographic Protocols, Adversaries, and Security Games

In this section we give a simple abstraction layer that can be used to specify cryptographic primitives, protocols, and security games. This is not meant to be a full high-level verification framework, but rather a useful abstraction for organizing proofs in our system. In future work it would be useful to implement a more robust abstraction to better automate game-hopping chains and indistinguishability arguments.

At a basic level, we represent cryptographic protocols as structures containing fields for each function in the protocol. A naive implementation of this however wouldn't be parametric in the set of oracles, forcing e.g. distinct definitions in/outside the random oracle model. To solve this we introduce a base structure `OracleAlg spec` that just contains a specification of how to simulate oracles in `spec` using `unifSpec`. Here `spec` should be thought of as the global oracles for a protocol (e.g. a random oracle), and the structure as containing the "intended behavior" of them. We also assume an intended initial state (e.g. an empty cache). Because we intend to use this for cryptographic protocols, we assume these are all indexed by some security parameter:

```
structure OracleAlg {ι : Type}
    (spec : ℕ → OracleSpec ι) where
  baseState (sp : ℕ) : Type
  init_state (sp : ℕ) : baseState sp
  baseSimOracle (sp : ℕ) :
    spec sp →[baseState sp] unifSpec
```

Given some `alg : OracleAlg spec` we will write `alg.exec oa` as shorthand for simulating `oa` with the bundled oracle in `alg`. This allows us to define the type of a crypto-system in a way that is agnostic to the oracles that are available. As an example we define the type of a signature protocol as extending this structure with keygen, sign, and verify functions:

```
structure SigAlg (spec : ℕ → OracleSpec ι)
    (M PK SK S : ℕ → Type)
    extends OracleAlg spec where
  keygen (sp : ℕ) :
    OracleComp (spec sp) (PK sp × SK sp)
  sign (sp : ℕ) : PK sp → SK sp →
    M sp → OracleComp (spec sp) (S sp)
  verify (sp : ℕ) : PK sp → M sp →
    S sp → OracleComp (spec sp) Bool
```

Any particular implementation is then required to specify how it intends the oracles to behave. We also use this to define a simple type to represent security games:

```
structure SecExp (spec : ℕ → OracleSpec ι)
    extends OracleAlg spec where
  main (sp : ℕ) : OracleComp (spec sp) Bool

def advantage (exp : SecExp spec) (n : ℕ) :=
  [= true | exp.exec n (exp.main n)]
```

For example soundness of a signature algorithm can be expressed in the following experiment, showing that for any message distribution `mDist` the signature algorithm produces a valid signature all but negligibly often:

```
variable (sigAlg : SigAlg spec M PK SK S)

def soundnessExp (mDist : (sp : ℕ) →
        OracleComp (spec sp) (M sp)) :
    SecExp spec where
  main := λ sp ↦ do
    let m ← mDist sp
    let (pk, sk) ← sigAlg.keygen sp
    let σ ← sigAlg.sign sp pk sk m
    sigAlg.verify sp pk m σ
  __ := sigAlg -- Inherit remaining fields

def isSound : Prop := ∀ mDist, negligible
  1 - (soundnessExp sigAlg mDist).advantage
```

The predicate `negligible` above is a special case of mathlib's more general definition of super-polynomial decay.

TODO: below

6.1. Adversaries and Asymptotics

For more complicated security games we need a representation of an adversary, and so we define a type `SecAdv spec α β` for an adversary that takes a value of type `α` and computes an output of type `β` using oracles in `spec`. We also require that the adversary is polynomial time [5], and that we have an explicit bound on the number of queries they make. Having a bound on the number of queries is essential to our eventual forking lemma, as the probability of successfully forking will depend on this count. For technical reasons we also require a list of the indices that have non-zero counts. Explicitly:

```
structure SecAdv (spec : OracleSpec)
    (α β : Type) where
  run : α → OracleComp spec β
  run_polyTime : polyTimeOracleComp run
  activeOracles : List spec.ι
  queryBound : spec.ι → ℕ
  queryBound_is_bound : ∀ count x y,
    ((y, count) ∈ support
      (simulate countingOracle (run x) 0)) →
      ∀ i : spec.ι, count i ≤ queryBound i
  activeOracles_eq : ∀ i,
    i ∈ activeOracles ↔ queryBound i ≠ 0
```

For example in the unforgeability experiment for a signature algorithm, we have an adversary that has access to the regular oracles `spec` of the protocol, and a signing oracle $M \to_{o} S$ that that allows them to query for a valid signature on any message. The input generation for this experiment is just the keygen function, and the main function gives the adversary the public keys and gets back a message `m` and signature `σ`. The experiment succeeds if this signature is valid and the message was never queried:

5. This definition is a fairly direct extension of a definition from mathlib based on Turing machines. We extend the definition to our monad in a natural way by requiring that any pure values returned are polynomial time, and that both computations in a bind are polynomial time. The extra variables introduced by bind are handled by currying

```
def unforgeableExp
    (adv : SecAdv (spec ++ M →ₒ S) PK (M × S)) :
    SecExp spec (PK × SK) where
  inpGen := sigAlg.keygen ()
  main := λ (pk, sk) ↦ do
    let ((m, σ), log) ←
      simulate (sigAlg.signingOracle pk sk)
        (adv.run pk) (emptyLog spec)
    let b ← sigAlg.verify pk m σ
    return (b && !(log.wasQueried m))
  __ := sigAlg
```

6.2. Hardness Assumptions

In this section we use the above framework to define an analogue of the discrete log problem, which we will use as the underlying hard problem for our signature scheme.

Our definition is based on hard homogeneous spaces as originally defined by Couveignes [15]. A homogenous space consists of a finite group of ”vectors” G acting simply and transitively on a set of base points P of the same cardinality. Equivalently this means the action of any element $g \in G$ is a bijection on P, and so for any pair of points there is a unique vector sending the first to the second, which is called the vectorization of the two points. We will write $g +_v p$ for the group action and $p_1 -_v p_2$ for vectorization.

The prototypical example is the discrete log problem in a cyclic group C of order n, where P is the generators of C and G is $(Z/nZ)^*$ acting by exponentiation, in which case the vectorization is exactly the discrete log. Another example is the ideal class group of a quadratic field acting on a certain class of elliptic curves, see the CSIDH protocol [16].

We define a type-class `HomogeneousSpace G P` to represent this in Lean, where `G` and `P` are types indexed by some security parameter. We can then define hardness of the vectorization problem via the following game, where the input generator just chooses two random points, the adversary attempts to find their vectorization, and then validation checks if this is correct:

```
variable (G P : (sp : ℕ) → Type)

def vectorizationAdv :=
  SecAdv (λ _ ↦ unifSpec)
    (λ sp ↦ P sp × P sp) (λ sp ↦ G sp)

def vectorizationExp [HomogeneousSpace G P]
    (adv : vectorizationAdv G P) :
    SecExp (λ _ ↦ unifSpec) where
  main := λ sp ↦ do
    let x₁ ←$ P sp
    let x₂ ←$ P sp
    let g ← adv.run sp (x₁, x₂)
    return g = x₁ -▯ x₂
  __ := baseOracleAlg
```

```
def vectorizationHard
    [HomogeneousSpace G P] : Prop :=
  ∀ adv : vectorizationAdv G P, negligible
    (vectorizationExp G P adv).advantage
```

The security of our signature scheme will be based on a reduction of an adversary that can forge signatures non-negligibly to one that can succeed at this experiment non-negligibly.

7. A General Forking Lemma

The forking lemma is a lemma commonly used in proving the security of signature schemes, which roughly states that if an adversary is able to, on inputs drawn from some distribution, non-negligibly compute a value with some property, then it's possible to construct an adversary that non-negligibly computes two such values on an input drawn from the same distribution, by running the adversary twice. Importantly though, these two executions of the adversary are guaranteed to line up until some specified point in the computation, at which point they diverge. The most common application is to fork the execution of an adversary at the result of a random oracle call, to get two distinct results corresponding to different outputs of the random oracle.

In this section we implement a version of the forking lemma based on the presentation given by by Bellare and Neven modified to align with our representation of computations. More explicitly, our forking lemma will be a formalization of the following:

Theorem 1 (Forking Lemma). Let IG be a probabilistic computation with return type α, A be an adversary that on an input from IG outputs a value of type β, and choose a particular oracle O from among those A has access to. Further, assume cf is a function that takes a value of type $\alpha \times \beta$ and chooses either an index i or aborts.
Then there exists an algorithm $F(A, cf)$ that on input x from IG produces two values y_1 and y_2, each of type β, such that:

- Both are possible outputs from executing A
- $cf\ (x, y_1)$ and $cf\ (x, y_2)$ are the same if they exist
- A's execution in both cases is identical until at least the nth query to oracle O
- The outputs of the nth queries to oracle O differ

Furthermore, if cf aborts only negligibly often on A then it also aborts only negligibly on $F(A, cf)$ (assuming that A makes at most a polynomial number of queries).

In this formulation the query to fork is given by cf as an index corresponding to how many queries should be replayed before allowing the computation to diverge. A priori this integer can't depend on an the actual input/output values of the oracle calls, it can only depend on the observed input and observed output of the adversary. When we later apply our forking lemma to the security of a signature scheme, we solve this by simulating the adversary with a logging oracle first, allowing us to internalize these values into the adversary's output. The index to fork on is then chosen by counting the position of the relevant query in the log.

7.1. Seeding Query Outputs

Our eventual implementation will depend on generating seeded values to use in answering oracle queries, preserving only a portion of the seed on the second execution of the algorithm. Importantly we must pregenerate values for all oracle calls, not just to the oracle we plan to fork, or else the adversary may diverge early before the desired query.

We represent such a seed as a function from an oracle index to a list of outputs for that oracle:

```
def QuerySeed (spec : OracleSpec) : Type :=
  (i : spec.ι) → List (spec.range i)
```

We could alternatively keep a single list where each seed is tagged with the index, but this leads to issues when actually using seeds in a computation. We write `∅` for the seed `λ _ ↦ []`. Operations like adding or removing a value are defined by continuation passing:

```
def dropAtIndex (qs : QuerySeed spec)
    (i : spec.ι) (n : ℕ) : QuerySeed spec :=
  λ j ↦ if j = i then (qs j).drop n else qs j
```

We then define a function `genSeed` for choosing random seed values, using a map `qc` to specify the number of values to generate. Because the set of oracles is allowed to be infinite, we also need to take in a list of oracles we should generate seeds for, to ensure the computation terminates. In practice both of these will be from the query bounds bundled into an adversary.

```
-- Helper function to perform the recursion
def genSeedAux (qc : spec.ι → ℕ) :
    List spec.ι → QuerySeed spec →
      OracleComp unifSpec (QuerySeed spec)
  | [], seed => return seed
  | j :: js, seed => do
      let xs ←$ Vector (spec.range j) (qc j)
      let seed' := seed.addValues xs.toList
      genSeedAux qc js seed'

def genSeed (qc : spec.ι → ℕ)
    (activeOracles : List spec.ι) :
    OracleComp unifSpec (QuerySeed spec) :=
  genSeedAux qc activeOracles ∅
```

It can be shown that the support of `genSeed` is the set of seeds for which the length of `seed i` is `qc i * k` for all `i`, where `k` is the number of times `i` appears in `activeOracles`. Furthermore, the output probability is uniform across these elements.

Once we have a seed, we can use a simulation oracle to substitute the seeded values in for the queries. Importantly, if we fail to find a seed value for a particular query we throw out the whole remaining seed, even if seed values might exist for other oracles. This ensures that computation will fully diverge after that point:

```
def seededOracle :
    spec →[QuerySeed spec] spec :=
  λ i t seed ↦ match seed i with
    -- value found: return it and update seed
    | u :: us => return (u, seed.update i us)
    -- no value: leave the query untouched
    | [] => (·, ∅) <$> query i t
```

It can be shown that for arbitrary values of the arguments to `genSeed`, running `genSeed` and simulating a computation with the result gives a computation that is identical to the original *up to distribution semantics*.

7.2. Constructing the Forking Algorithm

We are now ready to construct the algorithm $F(A, cf)$ as defined in the lemma above. We first extend the structure `SecAdv` in order to bundle in the function that chooses the forking point. To do this we include an additional argument `i` specifying which oracle will be forked, and include a function `cf` that returns the intended query index to fork at. We allow `cf` to optionally return `none`, and use `Fin` to enforce that the index is in bounds:

```
structure ForkAdv (spec : OracleSpec)
    (α β : Type) (i : spec.ι)
    extends SecAdv spec α β where
  cf : α → β → Option (Fin (qb i + 1))
```

Our forking algorithm for such an adversary is given in Figure 1. We start by choosing ahead of time an index `s` that we will fork on, and generate a seed with enough values for all oracles, except oracle `i` for which we generate only `s` values. We then add an additional value to each of the seeds at index `i`, and run the adversary separately with both seeds. Finally we check that the results both give `s` as the chosen fork point, and that the values added to the two seeds are different, returning a value only if both hold.

A concrete probability bound for this construction is given by the following theorem:

```
theorem le_fork_advantage
    (adv : ForkAdv spec α β i) (x : α) :
  let frk := [isSome | (fork adv).run x]
  let acc := [isSome | adv.cf x <$> adv.run x]
  -- Max number of queries by adversary
  let q : ℝ≥0∞ := adv.queryBound i + 1
  -- Number of possible hashes
  let h : ℝ≥0∞ := card (spec.range i)
  (acc / q) ^ 2 - acc / h ≤ frk := _
```

```
def fork (adv : ForkAdv spec α β i) :
    SecAdv spec α (Option (β × β)) where
  run := λ x ↦ do
    -- pre-select where to fork execution
    let s : Fin _ ← $[0..adv.queryBound i]
    -- Generate shared seed for both runs
    let qc := update adv.queryBound i s
    let seed : QuerySeed spec ←
        generateSeed qc adv.activeOracles
    -- Add the forked queries to the two seeds
    let seed₁ ← seed.insert i <$>
        ($ spec.range i)
    let seed₂ ← seed.insert i <$>
        ($ spec.range i)
    -- Run the adversary on both seeds
    let y₁ ← simulate' seededOracle
        seed₁ (adv.run x)
    let y₂ ← simulate' seededOracle
        seed₂ (adv.run x)
    -- Only return a value on success
    if adv.cf x y₁ = some s ∧
        adv.cf x y₂ = some s ∧
        (seed₁ i)[s] ≠ (seed₂ i)[s]
      then return some (y₁, y₂)
      else return none
  -- At most twice as many queries
  queryBound := 2 • adv.queryBound
  activeOracles := adv.activeOracles
  __ := _ -- Other fields omitted
```

Figure 1. Construction of the Forking Algorithm

The proof is omitted here, see the code for a full proof, however it essentially follows directly from Jensen's inequality combined with some low-level manipulation of summations. We note that this bound isn't as tight as the one given by in particular we have q^2 instead of q because we choose s immediately rather than waiting to see what value of s comes from the first execution of the computation.

8. A Fiat-Shamir Heuristic

In this section we give a definition of Σ-protocols, and give a version of the Fiat-Shamir Heuristic for constructing signature protocols for them. The type of such a protocol is given by the following structure, where `p` is the hard relation, `X` is the set of statements, `W` the type of witnesses to statements in `X`, `C` is the type of commitments, `Ω` is the type of challenges, and `P` is the type of proofs after commitment:

```
structure SigmaAlg {ι : Type}
    (spec : ℕ → OracleSpec ι)
    (X W C Ω P : ℕ → Type)
    (p : (sp : ℕ) → X sp → W sp → Prop)
    extends OracleAlg spec where
  commit (sp : ℕ) : X sp → W sp →
    OracleComp (spec sp) (C sp)
  prove (sp : ℕ) : X sp → W sp → C sp →
    Ω sp → OracleComp (spec sp) (P sp)
  verify (sp : ℕ) : X sp → Ω sp → P sp →
    OracleComp (spec sp) (C sp)
  sim (sp : ℕ) : X sp →
    OracleComp (spec sp) (C sp)
  extract (sp : ℕ) : P sp → P sp →
    OracleComp (spec sp) (W sp)
```

Our version of the Fiat-Shamir Heuristic then gives a way to construct a signature scheme from such a protocol given a key generation function `keygen` and a random oracle from messages and commitments to the challenge space:

```
def FiatShamir (M : ℕ → Type)
    (sigmaAlg : SigmaAlg spec X W C Ω P p) :
    SignatureAlg (spec := λ sp ↦
        spec sp ++ₒ (M sp × C sp →ₒ Ω sp))
      (M := M) (PK := X) (SK := W)
      (S := λ sp ↦ Ω sp × P sp) where
  keygen := λ sp ↦ keygen sp
  sign := λ sp pk sk m ↦ do
    let c ← sigmaAlg.commit sp pk sk
    let r ← query (Sum.inr ()) (m, c)
    let s ← sigmaAlg.prove sp pk sk c r
    return (r, s)
  verify := λ sp pk m (r, s) ↦ do
    let c ← sigmaAlg.verify sp pk r s
    let r' ← query (Sum.inr ()) (m, c)
    return r = r'
  baseState := λ sp ↦
    sigmaAlg.baseState sp × QueryCache _
  init_state := λ sp ↦
    (sigmaAlg.init_state sp, ∅)
  baseSimOracle := λ sp ↦
    sigmaAlg.baseSimOracle sp ++ randOracle
```

9. Related Work

Many different approaches to computer aided cryptography have been developed in order to verify cryptographic protocols in ways that can be checked by computer. In their recent survey of the field [17], Barbosa et al. point to three broad areas of focus: Design level security of protocols, functional correctness of implementations, and low-level side channel resistance. Our approach is fully within the scope of design level security, and so we restrict our discussion to that scope. Within this, they identify two main approaches: symbolic and computational.

In the symbolic approach messages (such as keys, nonces, etc.) are represented as atomic terms, and cryptographic primitives are modeled as black-box functions. An equational theory is used to represent the way these functions act on terms (for example an equational theory for symmetric encryption would include something like `Dec(Enc(m, k), k) = m`). Tools such as Tamarin [18] and ProVerif [19] can use these theories to automatically determine what an adversary could learn from the protocol. This supports significant automation but is necessarily constrained to the equational theory it is given, and failing to enumerate a full theory could lead to a vulnerability being missed (as happened to the verified SSH authenticated encryption scheme [2]). There are also issues of reusability due to lack of modularity in these reasoning systems, although some recent work has demonstrated some methods to solve this issue [10].

Computational models on the other hand treat all values as bit-strings, protocols as algorithms, and adversaries as Turing complete. This means that such proofs often give much more confidence in the real world security of verified protocols, however they typically suffer in terms of automation and scalability.

One approach to the computational model, and the one we take, is to embed protocols in some type of proof assistant or dependently typed language (Coq, Lean, Isabelle, etc.), and to then reason about the protocols within using the higher-order logic of the system. This is also the approach taken by FCF [7], CryptHOL [9], and EasyCrypt [20]. Of all the approaches this provides the greatest expressivity, in theory allowing essentially the same depth of reasoning as on-paper proofs. This approach allows for the greatest modularity and code reuse. However this expressiveness creates a higher proof burden, generally requiring the user to manually define cryptographic reductions for protocols. Although much work has been done to automate the verification of such reductions once they are defined, this still hinders automation compared to higher level approaches.

In contrast, CryptoVerif [5], Squirrel [21], and OWL [10] provide higher levels of automation while still remaining in the computational model. However CryptoVerif and Squirrel have minimal support for modularity, and this additional automation also comes at the cost of having a larger unverified code base in their foundations. One interesting avenue for future research could be to verify one of these frameworks in terms of another lower-level computational approach. There is also less support for very low-level manipulation of adversaries, and it's not clear that something like our forking lemma could be implemented non-axiomatically in such a system.

10. Conclusion

We present VCVio, a novel framework for verifying security proofs of cryptographic protocols, taking a foundational approach to representing and reasoning about their behavior. This framework is particularly powerful in its ability to represent and manipulate a computation's oracles, and we demonstrate this by implementing a much more general rewinding mechanism than in any previous work. Finally we use this to formally verify the security of a Schnorr-style signature, which to our knowledge hasn't been verified in any previous work.

References

[1] Q. Dao, J. Miller, O. Wright, and P. Grubbs, "Weak fiat-shamir attacks on modern proof systems," *2023 IEEE Symposium on Security and Privacy (SP)*, pp. 199–216, 2023.

[2] M. Bellare, T. Kohno, and C. Namprempre, "Breaking and provably repairing the ssh authenticated encryption scheme: A case study of the encode-then-encrypt-and- mac paradigm," *ACM Transactions on Information and System Security*, 2004.

[3] H. Krawczyk, "The order of encryption and authentication for protecting communications (or: How secure is ssl?)," *In Proceedings of IACR CRYPTO*, 2001.

[4] R. Canetti, A. Stoughton, and M. Varia, "Easyuc: Using easycrypt to mechanize proofs of universally composable security," *32nd IEEE Computer Security Foundations Symposium (CSF)*, 2019.

[5] B. Blanchet, "Cryptoverif: A computationally-sound security protocol verifier," 2017.

[6] P. G. Haselwarter, E. Rivas, A. V. Muylder, T. Winterhalter, C. Abate, N. Sidorenco, C. Hritcu, K. Maillard, and B. Spitters, "Ssprove: A foundational framework for modular cryptographic proofs in coq," 2021.

[7] A. Petcher, "A foundational proof framework for cryptography," *Doctoral dissertation, Harvard University, Graduate School of Arts & Sciences*, 2015.

[8] S. Zanella-Beguelin, "Formal certification of game-based cryptographic proofs," 2011.

[9] D. Basin, A. Lochbihler, and R. Sefidgar, "Crypthol: Game-based proofs in higher-order logic," 2019.

[10] J. Gancher, S. Gibson, P. Singh, S. Dharanikota, and B. Parno, "Owl: Compositional verification of security protocols via an information-flow type system," *In Proceedings of the IEEE Security and Privacy*, 2023.

[11] M. Bellare and G. Naven, "New multi-signature schemes and a general forking lemma," 2005.

[12] D. Firsoz and D. Unruh, "Reflection, rewinding, and coin-toss in easycrypt," *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2022.

[13] T. M. Community, "The lean mathematical library," in *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, January 2020.

[14] J. Lupo, "cryptolib: Security proofs in the lean theorem prover," *Master of Science Dissertation, University of Edinburgh*, 2021.

[15] J.-M. Couveignes, "Hard homogenous spaces," 2006.

[16] W. Castryck, T. Lange, C. Martindale, L. Panny, and J. Renes, "Csidh: An efficient post-quantum commutative group action," *ASIACRYPT*, no. LNCS 11274, pages 395–427, 2018.

[17] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao, and B. Parno, "Sok: Computer-aided cryptography," in *2021 IEEE Symposium on Security and Privacy (SP)*, pp. 777–795, 2021.

[18] S. Meier, B. Schmidt, C. Cremers, and D. A. Basin, "The tamarin prover for the symbolic analysis of security protocols," *In Proc. (CAV)*, 2013.

[19] B. Blanchet, "Modeling and verifying security protocols with the applied pi calculus and proverif," *Foundations and Trends in Privacy and Security*, 2016.

[20] G. Barthe, B. Gregoire, S. Heraud, and S. Zanella-Beguelin, "Computer-aided security proofs for the working cryptographer," *In Proceedings of IACR CRYPTO*, 2011.

[21] D. Baelde, S. Delaune, C. Jacomme, A. Koutsos, and S. Moreau, "An interactive prover for protocol verification in the computational model," *In Proceedings of the IEEE Security and Privacy*, 2021.