VCVio: Verified Cryptography in Lean via Oracle Effects and Handlers

Devon Tuma
University of Minnesota
Minneapolis, USA

Quang Dao
Carnegie Mellon University
Pittsburgh, USA

James Waters
Carnegie Mellon University
Pittsburgh, USA

Alexander Hicks
Ethereum Foundation
Edinburgh, UK

Nicholas Hopper
University of Minnesota
Minneapolis, USA

Abstract

Mechanized cryptographic proofs face a trade-off between assurance and expressiveness. Foundational frameworks, which reduce every proof step to the kernel of a general-purpose proof assistant, offer a small, auditable trusted base, but struggle to model the oracle manipulations and rewinding arguments pervasive in modern cryptography. They also lack the tactic infrastructure of specialized, non-foundational tools like EasyCrypt.

We present VCVio, a foundational Lean 4 framework that closes both gaps using established ideas from programming-language theory: algebraic effects and handlers for oracles, and a modular relational program logic for tactics. Concretely, a computation with oracle access is the free monad over the polynomial functor determined by its oracle specification, exposing its interaction history as an explicit syntax tree. Caching, logging, reprogramming, and seed pre-sampling become handler combinators; rewinding reduces to deterministic transcript replay without internal adversary state.

On top of the oracle core, VCVio provides two reusable layers. We extend the recent Loom framework (POPL 2026) to the relational setting, yielding a single tactic framework for both unary and relational probabilistic reasoning. Alongside this, our treatment of state-separating proofs achieves compositional separation by typing, whereas Nominal SSProve recovers it by quotienting locations modulo alpha equivalence.

We exercise this stack on three case studies: a random-oracle commitment scheme; the Bellare–Neven forking lemma, mechanized without the rewindability axioms used in the recent EasyCrypt formalization by Firsov and Janků; and the Schnorr signature scheme establishing EUF-CMA security. Much of our development used LLM coding agents and automated proof-search systems; we report on the workflows, successes, and failure modes as a data point in LLM-assisted theorem proving.

Keywords

formal verification, Lean, game-based cryptography, oracle semantics, Fiat–Shamir, forking lemma

1 Introduction

Formal verification of cryptography has matured into an end-to-end pipeline, with machine-checked security proofs at one end [ZB11, Pet15, BLS19, HRM+21, Spi26] and verified compilation to architecture-specific assembly at the other [HHH+24, BDF+26]. On the proof side, *foundational* frameworks (those reducing every step to the kernel of a general-purpose proof assistant) typically pay for their small trusted base with more restrictive oracle models, less expressive tactic libraries, and a smaller stock of reusable proof infrastructure than non-foundational tools such as EasyCrypt [BGHZB11]. This paper asks how to close that gap on all three axes at once.

The choice of oracle representation has far-reaching consequences for which proof techniques are natively expressible. FCF [Pet15] restricts oracle access to a single oracle with fixed input and output types, making reasoning about multiple simultaneous oracles awkward. CryptHOL [BLS19] represents games as generative probabilistic values (GPVs),[1] supporting equational reasoning but not exposing adversary interactions as replayable syntax. SSProve [HRM+21] uses a heap-passing sub-distribution monad that models oracle state naturally but requires explicit state-separation machinery, recently addressed by nominal techniques [LS25]. CatCrypt [Spi26], a recent Lean development in the same tradition, adds native forking-lemma infrastructure but retains the heap/package model. EasyCrypt [BGHZB11], which historically offered the broadest tactic vocabulary of the systems above but is not foundational, supports rewinding through an additional probabilistic-reflection layer introducing several new axioms [FU22, FJ25]. Table 1 summarises how each system compares on the criteria that matter for the proofs in this paper.

We present **VCVio**, a foundational framework implemented in Lean 4 designed around an explicit syntactic representation of oracle interactions: an adversary's queries to its oracles, and the responses it receives, are exposed as a syntax tree the framework can replay, reinterpret, and compose, rather than evaluated away into an opaque (sub-)probability distribution. The underlying machinery (interaction trees [XZH+19, yXZH+19] and their inductive variant) is a well-established architecture for modeling effectful programs, with adoption in verified compilation [ZNMZ12], compiler-security preservation [AOBB+25], layered interpreters [YZZ22], higher-order and Isabelle/HOL interaction-tree developments [AS25, FHW25], and effect-generic program logics [KYW24, GPZ+26, The26b]. On top of this representation, we build the rest of the paper as four layers sharing the same oracle computation foundation: oracle handlers, the program logic, the tactic infrastructure, and the state-separating proof layer.

1.1 Our Contributions

An algebraic-effects core for oracle computations. We give a dictionary (Table 2) between the everyday vocabulary of cryptographic

[1] A coinductive resumption monad over sub-probability mass functions.

Tool	Host	Foundational	Open source	Oracle / game model	End-to-end EUF-CMA	Relational reasoning
CertiCrypt [ZB11]	Rocq	yes	yes	deeply-embedded pWHILE	not mechanized	pRHL
EasyCrypt [BGHZB11]	standalone	no	yes	pWHILE + shared memory	yes, via 11 axioms [FU22, FJ25]	pRHL, eRHL, prob. reflection
FCF [Pet15]	Rocq	yes	yes	one oracle / game, fixed I/O	not mechanized	n/a
CryptHOL [BLS19]	Isabelle/HOL	yes	yes	GPV (coinductive resumption monad)	not mechanized	equational SPMF reasoning
SSProve [HRM+21]	Rocq	yes	yes	heap-passing SPMF + SSP packages	not mechanized	invariant-based RHL
CatCrypt [Spi26]	Lean	yes	no†	SSProve port (same model)	not mechanized	inherited from SSProve
VCVio	**Lean**	**yes**	**yes**	**free monad over polynomial functor + handlers for interpretation**	**yes, no rewindability axioms (Section 8.2.4)**	**pRHL + eRHL via relational ordered monad algebra (Section 6)**

Table 1: Computational frameworks for general-purpose game-based cryptography. The choice of oracle representation determines which end-to-end proofs are natively expressible: EasyCrypt and VCVio both support Fiat–Shamir-style EUF-CMA proofs through a Bellare–Neven-style forking lemma [BN06], and only VCVio does so without rewindability axioms. † As of this submission, CatCrypt's repository is not publicly available.

oracle reasoning and the well-developed theory of algebraic effects [PP03, PP09], handlers, and free monads over polynomial functors.

Standard oracle manipulations (caching, logging, programming, seed pre-sampling, rewinding) become composable handler combinators rather than ad hoc constructions, and probabilistic sampling is a query to a uniform-selection oracle, unifying the probabilistic and oracle-interaction layers. Embedding a smaller oracle interface into a larger one is a structured map between polynomial functors (a *lens*, Definition 3.1); a mild side condition (cartesianness) makes the embedding probability-preserving (Theorem 5.1), so oracle coercions can be elaborated automatically without disturbing the program logic. We build on Lean and Mathlib [mC20], which provides a community-maintained library for probability theory, algebraic structures, and polynomial functors.

A modular probabilistic and relational program logic. On top of the oracle core, we build a two-mode program logic (Section 6): a quantitative unary mode for expectation and probability bounds, and a relational mode for distributional couplings between two computations possibly carrying different oracle handler stacks. We obtain both modes by extending the recent Loom framework [GPZ+26] of *ordered monad algebras* from the unary to the relational setting (Definition 6.2); its quantitative specialization gives the predicate-transformer counterpart of Avanzini et al.'s eRHL [ABDG25]. The resulting algebra is parametric in the underlying monadic stack and matches the expressive power of both the *simple* and *generic* frameworks of Maillard et al.'s *Next 700 Relational Program Logics* [MHRV20]. The logic comes with two tactics that we design, `vcgen` and `rvcgen`, plus a port of EasyCrypt's relational tactic vocabulary (Table 3); together they extend the Lean standard library's `mvcgen` [The26b] and allow us to better reason about game-hopping arguments.

State-separating proofs with separation by typing. We recast Brzuska et al.'s state-separating proofs [BDLF+18] as a layer over the oracle core (Section 7): a package is a stateful query handler, package composition is handler composition, and the SSP reduction lemma becomes a propositional equality of oracle computations rather than a meta-theorem stated at the level of adversary advantages. State separation is forced by the handler's type, instead of being enforced as a side condition on a shared heap, giving compositional separation that SSProve [HRM+21] and Nominal SSProve [LS25] obtain through dedicated separation side conditions (see Section 7 for the comparison). Every tool from the program logic (`vcgen`, `rvcgen`, the EasyCrypt-style vocabulary, the coupling rules) applies inside a package proof without translation, since handlers and the program logic share the same underlying oracle computation syntax. We exercise the layer on the Fiat–Shamir / Schnorr case study below, structuring the chosen-message-to-no-message-attack reduction as a chain of state-separating game hops over a packaged random oracle (Section 8.2.4).

Three end-to-end case studies, without domain-specific axioms. We give, to our knowledge, the first *foundational* mechanizations of the Bellare–Neven forking lemma [BN06] and of EUF-CMA security for the Schnorr signature scheme [Sch91] (obtained from the Schnorr identification protocol via the Fiat–Shamir transform). We also formalize a textbook random-oracle commitment scheme [CY24, Part IV] that exercises the common handler toolkit (Section 8). Unlike the recent EasyCrypt formalization of Firsov and Janků [FJ25], the closest precedent on the forking-lemma side, our proofs require no additional axioms about adversary state, no decomposition of the adversary into stateful phases (their `init`/`continue`/`finish` split), and no auxiliary axioms for bitstring pairing; the analogue of their save-run-restore axiom is a distributional coupling we prove inside the program logic by induction on the adversary's syntax tree. The Fiat–Shamir reduction is structured as a chain of state-separating game hops over the package layer of Section 7, and instantiates to Schnorr by discharging three properties of the underlying Σ-protocol (special soundness, perfect HVZK, commit-predictability), recovering the textbook Pointcheval–Stern reduction [PS00].

Implementation and adoption. VCVio is a Lean 4 library of over 90,000 lines built on Mathlib [mC20], with new tactics extending the Lean standard library and a body of new theory on polynomial functors, free monads, and relational monad algebras staged for upstreaming to Mathlib or Cslib [BCM+26]. The full repository is available on Github. The library has been adopted as a base layer in ArkLib's ongoing formalization of interactive oracle reductions [The26a] and in Dziembowski et al.'s Lean formalization

of computationally sound symbolic cryptography [DFMS25], suggesting that the design generalizes beyond standard game-based security.

1.2 Organization

Section 2 reviews the Lean and monadic background. Sections 3 to 5 introduce oracle computations, effect handlers, and the denotational semantics. Section 6 develops the unary and relational program logics and their tactic support. Section 7 describes the state-separating package layer; the asymptotic-game machinery used by the case studies lives in Appendix D. Section 8 presents the commitment, forking-lemma, and Fiat–Shamir case studies, with the remaining case study in Appendix G. Sections 9 and 10 and Appendix A discuss related work, future directions, and our AI-assisted development workflow.

2 The Lean Proof Assistant

We implement our framework in Lean 4, a dependently typed functional language and interactive proof assistant whose syntax is similar to Haskell or OCaml. At the core is the calculus of constructions: every expression is a term with a type, denoted `(a : A)`. Functions are written `fun x => e`, and dependent types let argument and return types refer to earlier arguments; implicit arguments `{A : Type}` are filled in by type inference. Standard inductive types (`A × B`, `A ⊕ B`, `Unit`, `⊥`, `Bool`, `List`, `Option`) are built in. We make heavy use of the `mathlib` [mC20] library, which provides the mathematical foundations for our definitions and proofs. Much of the reusable theory developed for VCVio is staged for contribution back to Mathlib or Cslib [BCM+26].

Proofs may be written directly as terms in the language or interactively via *tactics*, a form of metaprogramming that allows users to write proofs in a stepwise-fashion. This metaprogramming is highly extensible, and new tactics can be defined by users, as we often do. Tactics range from primitive proof steps (`contradiction`, `cases`) to high-level decision procedures (`linarith`, `omega`, `grind`).

Monads and transformers. We represent computations with oracles via a monadic embedding, a common way to model side effects in pure functional languages. In Lean, a function `m : Type → Type` is a monad if we have operations `pure : A → m A` and `bind : m A → (A → m B) → m B` for returning values and sequencing computations respectively. The operation `bind mx my` is usually written `mx >>= my`. The standard monad laws (associativity of bind, left and right identity of pure) are captured by the `LawfulMonad m` type-class, which we implicitly assume going forward.

Monad transformers augment a base monad `m` with additional operations: `OptionT m` adds failure, `StateT σ m` adds mutable state of type σ, and `WriterT ω m` adds an output stream of type ω. Lean's `MonadLift m n` typeclass enables automatic lifting of computations from `m` into a richer monad `n`, analogous to coercions in mathematics: when a term in `m` appears where an `n`-computation is expected, Lean inserts the lift automatically. We assume basic familiarity with monadic programming; the appendix expands on the specific transformer stacks used by VCVio.

Cryptographic concept	Polynomial-functor / effects analogue
oracles available in a game	polynomial functor $P = (A_P, B_P)$
all possible queries	positions A_P
responses to a query t	fiber $B_P(t)$
algorithm with oracle access	free monad on P at α
algorithm with no queries	pure leaf α
a query, then a continuation	roll node $t \in A_P$ with children $B_P(t)$
oracle behavior (real, lazy, programmed, …)	P-algebra in m, i.e. $\Pi a.\, m\, B_P(a)$
running under a specific oracle	morphism of free monads
embedding a smaller oracle interface	lens $\ell : P_{spec} \to P_{spec'}$ (Definition 3.1)
embedding that preserves distributions	cartesian lens (Definition 3.3)
uniform random sampling	query to a uniform-selection oracle
lazy RO, query logging, programming	combinators via StateT, WriterT, ReaderT
rewinding the adversary	deterministic replay of the transcript

Table 2: Dictionary between cryptographic concepts (left) and their polynomial-functor / algebraic-effects analogues (right); Sections 3–5 work through each row.

3 Computations With Oracle Access

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 [Pet15]. Our monad is a unified generalization of the two monads used in that system. Alternatively, it can be seen as a specialized case of algebraic effects and handlers [PP09], with queries as events and implementations as event handlers. Table 2 summarizes the dictionary between the cryptographic and polynomial-functor / algebraic-effects vocabularies.

3.1 Specifying Oracles

We start by representing the set of oracles available to a computation as a polynomial functor [AAG05, GK13, NS25], represented in Lean by the type:

```
structure PFunctor : Type (max u v) where
  A : Type u
  B : A → Type v
```

The head type `A` will represent the set of possible oracle inputs for the computation, and the map `B` gives the result type of the corresponding query. Because we often want the head type to be exposed at the type level outside the structure, we define the type `OracleSpec A` to be simply the type of functions from `A` to `Type v`, equivalently the set of polynomial functors with head type `A`, and write `spec.toPFunctor` for the bundled `⟨A, spec⟩`.

Note that we don't specify any behavior for these oracles: this should be seen as analogous to a type signature for the set of oracles that can be queried.

We write `T →ₒ U` for a specification with input type `T` and constant output type `U`. For probabilistic computation, we use `coinSpec`

(a coin-flipping oracle returning `Bool`) or `unifSpec` (uniform selection over $[0, n]$ for any $n \in \mathbb{N}$). While it is possible to approximate uniform selection using only coin flips, it's simpler to use `unifSpec`, and we will default to that going forward.

3.2 Representing Computations

We then define a model of computations with oracle access as a type `OracleComp spec A`, where `spec` specifies what oracles the computation can make use of, and `A` 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.

As previewed in the introduction, we define `OracleComp spec` to be the free monad over `spec.toPFunctor` [Koc11]. We omit the full construction here; Appendix B records the precise Lean definitions. Computations in the resulting representation have exactly one of two canonical forms:

- `return x`
- `query_bind t cont`

Concretely, a computation either returns a pure value `x` in the output type, or makes a query on input `t` in the specification's head type and calls some continuation `cont` on the result (itself having access to the same oracle set). We write `(query t)` for the case where `cont` is just `return`. This type is naturally a monad, with the general bind being defined by pattern matching on the first computation. Pure values plug in directly, and for bind we have:

```
query_bind t cont >>= my =
    query_bind t (cont · >>= my)
```

This monad is lawful, and it is easy to show via the standard monad laws that:

```
query_bind t cont = do let u ← query t; cont u
```

We write `coin` and `$[0..n]` for querying `coinSpec` or `unifSpec` respectively, and write `ProbComp A` when the spec is `unifSpec`, i.e. a computation with no oracles besides uniform random selection. We use Lean's monad syntax and its associated notation to write computations; in particular we use `return a` for the pure operation, `oa >>= ob` for the bind operation, and use *do*-notation for sequencing larger computations:

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

Using these, we can 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 write `$ xs` for random selection from any collection. We can also define computations recursively via pattern matching, assuming the recursion used is well-founded.[2]

[2]Like most proof verification frameworks, Lean is limited to using well-founded recursion, ensuring that all recursive functions eventually terminate. Generally cryptographic protocols avoid using unbounded recursion so this is rarely an issue. Other formalizations have added a fixed-point operator to their syntax to solve this, but this comes at the cost of increased proof complexity.

3.3 Sub-Specs and Type Coercions

To combine sets of oracles, we equip `OracleSpec` with an addition operation `spec + spec'` that represents access to oracles in either of two original specs. The input set is a sum type of the original sets, using pattern matching to route queries to the first and second specs, respectively. This operation is neither commutative nor associative: swapping the order does not change the set of available oracles, but it changes how queries are addressed.

To bind a computation defined over one spec inside an ambient computation over a richer spec, we need an embedding $spec \subset_o spec'$ that translates queries on the left into queries on the right and translates the resulting responses back. This is exactly the data of a polynomial-functor lens [NS25].

Definition 3.1. A *lens* $\ell : P \to Q$ between polynomial functors is a pair of a forward map on positions and a fiberwise backward map on directions,

$$\ell^\sharp : A_P \to A_Q, \qquad \ell^\flat : (a : A_P) \to B_Q(\ell^\sharp\, a) \to B_P(a).$$

Lenses compose: $(\ell_2 \circ \ell_1)^\sharp = \ell_2^\sharp \circ \ell_1^\sharp$ on positions, and $(\ell_2 \circ \ell_1)^\flat\, a = \ell_1^\flat\, a \circ \ell_2^\flat\, (\ell_1^\sharp\, a)$ on directions.

Definition 3.2 (Sub-spec embedding). A `SubSpec` instance $spec \subset_o spec'$ is a lens $\ell : P_{spec} \to P_{spec'}$ together with the `MonadLift (OracleQuery spec) (OracleQuery spec')` action it induces by Yoneda,

$$\ell \cdot \langle t, k \rangle = \left\langle \ell^\sharp\, t,\ \ k \circ \ell^\flat\, t \right\rangle.$$

Composition of sub-specs is composition of the underlying lenses.

The basic embeddings $spec_1 \subset_o spec_1 + spec_2$ and $spec_2 \subset_o spec_1 + spec_2$ have $\ell^\sharp \in \{\mathsf{Sum.inl}, \mathsf{Sum.inr}\}$ and $\ell^\flat$ the identity on each fiber. Compared with the *subevent* typeclass of Xia et al. [yXZH+19], which presents event inclusion as a polymorphic injection $E\,\alpha \to F\,\alpha$, the lens presentation makes the polynomial structure explicit.

Lawful sub-specs. Not every sub-spec embedding is well-behaved on the probabilistic content of a query: an arbitrary backward map can collapse or duplicate fibers and distort the uniform distribution carried along. The right correctness condition is the classical one for polynomial functors.

Definition 3.3 (Cartesian lens). A lens $\ell : P \to Q$ is *cartesian* when every backward map $\ell^\flat\, a : B_Q(\ell^\sharp\, a) \to B_P(a)$ is a bijection. A `LawfulSubSpec` instance is a sub-spec embedding whose underlying lens is cartesian.

Cartesianness is strictly weaker than being a lens equivalence: the basic embeddings `Sum.inl`, `Sum.inr` above are cartesian (the backward maps are identities) but not equivalences (the forward map on positions is not surjective). Cartesianness, but not equivalence, is the structural property we need for coercions to interact correctly with probabilities; we make this precise in Theorem 5.1, once the denotational semantics are in place.

4 Implementing Oracle Behavior

The main semantics for `OracleComp` use `simulateQ`, which recursively substitutes oracle queries with some user-specified implementation. For example, simulating a random oracle will consist of

first substituting queries to cache and reuse previous query outputs, then further substituting the new queries with uniform responses.

Given some set of oracles `spec` to implement and a new monad `m` where the substituted computations will take place, a `QueryHandler spec m`[3] is an effect handler for `spec.toPFunctor` in the monad `m`, equivalently a function mapping each oracle input to an output computation in `m`:

```
def QueryHandler (spec : OracleSpec ι)
    (m : Type u → Type v) : Type _ :=
  (t : spec.Domain) → m (spec.Range t)
```

The function `simulateQ` performs this substitution by recursively traversing the computation, replacing each query with its implementation. Since `OracleComp` is a free monad, this mapping is the canonical *monadic extension* of the implementation, preserving pure values and sequencing for natural compatibility with monadic operations:

```
example : simulateQ impl (oa >>= ob) =
    simulateQ impl oa >>= (simulateQ impl ∘ ob) := _
example : simulateQ impl (f <$> oa) =
    f <$> simulateQ impl oa := _
```

As a simple example, we have an identity implementation that substitutes back the original query, using `OracleComp spec` itself as the output monad:

```
def QueryHandler.id :
    QueryHandler spec (OracleComp spec) := query
```

Similarly we can replaces oracle queries with a uniform choice of outputs, reducing an arbitrary set of oracles to a probabilistic computation by sampling uniformly over the output type:

```
def uniformSampleImpl
    [∀ t, SampleableType (spec.Range t)] :
    QueryHandler spec ProbComp := (fun t => $ᵗ spec.Range t)
```

We build implementations of complex oracles modularly, depending on the specification's structure. For example, given a combined specification $\texttt{spec}_1 + \texttt{spec}_2$ and implementations in monads $\texttt{m}_1$ and $\texttt{m}_2$, we obtain an implementation in a larger monad `n` assuming instances to lift the results:

```
def append {m₁ m₂ n : Type u → Type v}
    [MonadLift m₁ n] [MonadLift m₂ n]
    (impl₁ : QueryHandler spec₁ m₁)
    (impl₂ : QueryHandler spec₂ m₂) :
    QueryHandler (spec₁ + spec₂) n
  | .inl t => impl₁ t | .inr t => impl₂ t
```

4.1 Modular Implementation Extensions

Besides just implementing oracle behavior, another important use case is tracking some information about a computation. We implement this type of behavior as transformations to existing query implementations, applying monad transformers to the output monads. For example, we can add logging to an implementation by using the `WriterT` transformer to write to an output stream:

```
def withLogging (so : QueryHandler spec m) :
    QueryHandler spec (WriterT (QueryLog spec) m) :=
  fun t => do let u ← so t; tell [⟨t, u⟩]; return u
```

[3]The Lean formalization names this type `QueryImpl`; we write `QueryHandler` throughout the paper to align with the algebraic-effects vocabulary.

A specialization `countingOracle` replaces the full log with per-oracle counts, supporting a predicate `IsPerIndexQueryBound oa qb` that bounds queries to each oracle `t` by `qb t` (with a polynomial in the security parameter analogue `PolyQueries`; see Appendix D.3).

We can define a similar implementation that returns cached values on repeated inputs, and only forwards fresh queries to the base implementation, using `StateT` over `WriterT` as we need to both read and write to state.

We can then implement random oracles as:

```
example [∀ t, SampleableType (spec.Range t)] :
    QueryHandler spec (StateT (QueryCache spec) m) :=
  uniformSampleImpl.withCaching
```

We note that the ordering of these sorts of transformations can interact subtly: `so.withCaching.withLogging` will only log fresh queries, while `so.withLogging.withCaching` will log repeated queries of the same input.

5 Denotational Semantics

We give denotational semantics to computations via a typeclass `HasEvalSPMF` that maps a monad into sub-probability mass functions (`SPMF`). An `SPMF α` assigns to each `x : α` a probability in $\mathbb{R}_{\geq 0\infty}$ that may sum to less than 1 (accounting for possible non-termination or failure). Since `OracleComp` is a free monad, we can define its semantics by specifying the distribution for each query (uniform over the output type) and extending via the monad structure. We define three probability notations for a computation `mx`:

- `Pr[= x | mx]` - chance of output `x`
- `Pr[ p | mx]` - chance of event `p`
- `Pr[⊥ | mx]` - chance of failure

which satisfy the expected characterization:

```
Pr[=x | pure a] = if x = a then 1 else 0
Pr[=y | oa >>= ob] = ∑ x, Pr[=x | oa] * Pr[=y | ob x]
Pr[=u | query t] = 1 / Fintype.card (spec t)
```

Note that while `OracleComp` always has zero failure probability, computations with `OptionT` or `ErrorT` transformers on top may not. The summation in the bind case is a potentially infinite sum which can lead to issues regarding convergence if probabilities are taken to be real numbers, so we instead use the type $\mathbb{R}_{\geq 0\infty}$ of non-negative (potentially infinite) reals.

We can also write the probability of an event as a sum if `p` is decidable, giving:

```
Pr[p | mx] = ∑ x, if p x then Pr[= x | mx] else 0
```

Finally, it is possible to define a finite version of `support` for a computation using Lean's `Finset` type, which is useful in many proofs but requires that the output type have decidable equality.

Coercions and lifting. Definition 3.3 promised that lawful subspecs preserve probabilistic content; we now state this precisely.

Proposition 5.1 (Lifting preserves probability). *If* $\texttt{spec} \subset_o \texttt{spec}'$ *is lawful, then* $\texttt{evalDist}(\texttt{liftComp}\ c\ \texttt{spec}') = \texttt{evalDist}\ c$ *for every* $c : \texttt{OracleComp}\ \texttt{spec}\ \alpha$.

The proof is a structural induction on c; see Appendix C.1. Conversely, any non-cartesian sub-spec breaks Theorem 5.1 on some query, so cartesianness is exactly the structural property that makes

coercion probability-preserving. Every lemma of the form "the probability of an event in a coerced computation equals the probability in the original" is a downstream consequence; the program logic of Section 6 therefore suppresses mention of coercions automatically.

6 Program Logic

Game-based security proofs follow a structured pattern: rewrite a starting game through a sequence of distributionally close intermediate games, then bound the probability of an event in the final game. Manipulating `evalDist` directly does not scale for larger reductions; the standard tool in computer-aided cryptography is a *program logic* that internalises this style of reasoning [ZB11, BGHZB11, HRM+21, BLS19].

Our version is a two-mode program logic on top of `OracleComp`: a *quantitative unary* mode that bounds expectations and probabilities (Section 6.2), and a *relational* mode that proves game equivalences by exhibiting couplings (Section 6.3). Both modes are required to compose with the handler stacks of Section 3 (caching, logging, reprogramming, seeded replay) in arbitrary combinations on either side of a triple.

We achieve this by extending the recent algebraic weakest-precondition (WP) framework of Loom [GPZ+26] (Section 6.1) to a relational setting where the two sides of a triple may carry different transformer stacks (Section 6.3), and by building a tactic suite (Section 6.4) that extends the Lean standard library's `mvcgen` [The26b] with quantitative and relational generators and ports a library of EasyCrypt-style game-hopping tactics.

6.1 An Algebraic Core for Modular WPs

Loom [GPZ+26] observes that essentially every weakest-precondition calculus over a monad M arises from an *ordered monad algebra* $\mu : M\,L \to L$ on a complete lattice L, monotone in its argument and satisfying $\mu\,(\mathsf{return}\ \ l) = l$ and $\mu\big(c \gg\!= (\mu \circ f)\big) = \mu\big(c \gg\!= f\big)$ (Definition 6.1). The induced weakest precondition $\mathsf{wp}\ c\ \mathit{post} := \mu\big(c \gg\!= \mathsf{return} \circ \mathit{post}\big)$ defines Hoare triples $\{\mathit{pre}\}\ c\ \{\mathit{post}\} \ :\Leftrightarrow\ \mathit{pre} \le \mathsf{wp}\ c\ \mathit{post}$, and the standard structural rules (return, bind, consequence, frame) follow from the algebra laws.

The payoff is that monad transformers compose for free: Loom proves once and for all that an algebra on M lifts canonically through every standard transformer (StateT σ, ReaderT ρ, ExceptT ε, OptionT, non-determinism), with the lifted algebra valued in L (or an exponential of L for stateful layers). For `OracleComp` we instantiate the framework twice: with $L = \mathsf{Prop}$ and $\mu\,c\,P := \forall x \in \mathrm{support}\,c.\ P\,x$ for an almost-sure unary logic that bridges to the Lean standard library's `mvcgen` [The26b] via its `Std.Do.WP` class, and with $L = \mathbb{R}^{\infty}_{\ge 0}$ and $\mu\,c\,f := \sum_x \Pr[c = x] \cdot f\,x$ for the quantitative logic of Section 6.2 that states cryptographic advantage bounds. The same handler stack supports both verification modes without rebuilding the transformer machinery for either carrier.

What Loom does not provide is a relational logic, in which one program is related to another rather than to a postcondition. Closing this gap while respecting the heterogeneous transformer stacks of cryptographic reductions is the subject of Section 6.3.

Definition 6.1 (Ordered monad algebra, after Loom [GPZ+26]). Let M be a monad and L a complete lattice. An *ordered monad algebra* on (M, L) is a map $\mu : M\,L \to L$ that is monotone in its argument under the pointwise order on $M\,L$ and satisfies, for all α, $c : M\,\alpha$, and $f : \alpha \to M\,L$,

$$\mu\,(\mathsf{return}\ l) \ = \ l, \qquad \mu\big(c \gg\!= (\mu \circ f)\big) \ = \ \mu\big(c \gg\!= f\big).$$

6.2 Quantitative Unary Reasoning

Instantiating Definition 6.1 on `OracleComp` with $L = \mathbb{R}^{\infty}_{\ge 0}$ and μ taking expectation under `evalDist` yields

$$\mathsf{wp}\ c\ \mathit{post} \ = \ \sum_{x : \alpha} \Pr[c = x] \cdot \mathit{post}\,x \ = \ \mathbb{E}_{x \sim \mathsf{evalDist}\,c}\big[\mathit{post}\,x\big],$$

the carrier needed for cryptographic advantage bounds. The four standard structural rules (return, bind, consequence, frame) come from Definition 6.1 for free; the quantitative bind rule $\mathsf{wp}\,(c \gg\!= f)\ \mathit{post} \ = \ \mathsf{wp}\,c\,(\lambda a.\,\mathsf{wp}\,(f\,a)\ \mathit{post})$ is a one-line consequence of Tonelli on the discrete sum. The bridge to event probabilities is immediate: $\Pr[p \mid c] = \mathsf{wp}\,c\,(\mathbf{1}_p)$ for any predicate p (with $\mathbf{1}_p\,x = 1$ if $p\,x$ else 0), letting the unary tactic suite of Section 6.4 reduce probability-bound goals to weakest-precondition obligations.

6.3 Relational Reasoning over Heterogeneous Stacks

Game-based proofs routinely compare programs over potentially *different* oracle interfaces: the left game might cache its random oracle while the right logs it, and we want to relate their output distributions through some relational postcondition. This moves from a single algebra $\mu : M\,L \to L$ to a joint operator rwp taking one computation per side and a relational postcondition $\alpha \to \beta \to L$.

Definition 6.2 (Relational ordered monad algebra). Let M_1, M_2 be monads and L a complete lattice. A *relational ordered monad algebra* on (M_1, M_2, L) is an operator

$$\mathsf{rwp} : M_1\,\alpha \to M_2\,\beta \to (\alpha \to \beta \to L) \to L$$

satisfying $\mathsf{rwp}\,(\mathsf{pure}\,a)\,(\mathsf{pure}\,b)\ \mathit{post} \ = \ \mathit{post}\,a\,b$, monotonicity in *post*, and the lax bind law

$$\begin{aligned}\mathsf{rwp}\,c_1\,c_2\,\big(\lambda a\,b.\,\mathsf{rwp}\,(f\,a)\,(g\,b)\ \mathit{post}\big)\\ \le\ \mathsf{rwp}\,(c_1 \gg\!= f)\,(c_2 \gg\!= g)\ \mathit{post}.\end{aligned}$$

A *relational Hoare triple* $\{\mathit{pre}\}\ c_1 \sim c_2\ \{\mathit{post}\}$ asserts

$$\mathit{pre} \le \mathsf{rwp}\,c_1\,c_2\ \mathit{post}.$$

The lax bind is the relational effect observation of Maillard et al. [MHRV20].[4]

Two carriers, again. We instantiate Definition 6.2 on $(M_1, M_2) = (\mathsf{OracleComp}\ \mathit{spec}_1, \mathsf{OracleComp}\ \mathit{spec}_2)$ for the same two carriers as the unary case, with γ ranging over couplings of (c_1, c_2) (joint distributions on $\alpha \times \beta$ with the right marginals):

[4] It is in fact an equality whenever the underlying relational specification monad is deterministic in both arguments. We track this stronger property with a `StrictBind` subclass and use it where it holds (every `StateT` side-lift, all deterministic specifications); the lax form is what we need for `OracleComp` because the optimal coupling of a composite computation can be more precise than the sequential composition of optimal couplings.

- *Qualitative* ($L = \mathsf{Prop}$):

$$\mathsf{rwp}\, c_1\, c_2\, R := \exists\, \gamma.\ \forall (x, y) \in \mathsf{support}\, \gamma,\ \ R\, x\, y.$$

This is the existing `CouplingPost` predicate of VCVio's earlier program logic; the resulting triple reduces to the standard pRHL coupling-based triple [ZB11, BGHZB11, HRM+21].

- *Quantitative* ($L = \mathbb{R}^{\infty}_{\geq 0}$):

$$\mathsf{rwp}\, c_1\, c_2\, \mathit{post} := \inf_{\gamma} \sum_{(x,y)} \gamma(x, y) \cdot \mathit{post}\, x\, y.$$

This is the predicate-transformer counterpart of Avanzini et al.'s eRHL [ABDG25]: every eRHL judgment is equivalent to such an rwp-triple.

Specializing the qualitative carrier to $R = \mathsf{Eq}$ recovers distributional equality, and the quantitative one specialised to indicators recovers statistical distance, so a proof that two games produce identical output distributions is a coupling on the diagonal, built incrementally with the bind rule of Definition 6.2.

Heterogeneous transformer stacks. Relational reasoning makes transformer lifting genuinely two-sided. Maillard et al.'s framework already permits the two computations in a judgment to live in arbitrary, even distinct, monads [MHRV20]; our contribution is a reusable algebraic lifting story for transformer stacks over an existing rwp-algebra, including one-sided lifts that add, change, or drop a transformer on only one side. A reduction may move from a left game caching its random oracle (sitting in StateT (OracleCache) (OracleComp *spec*)) to a right game logging it (a different StateT), or even drop a transformer entirely on one side. We call these *heterogeneous side lifts*, and provide them for every standard transformer over any rwp-algebra: for state, the left-lift turns an algebra on (M_1, M_2, L) into one on $(\mathsf{StateT}\ \sigma\ M_1, M_2, \sigma \to L)$, with analogous right- and two-sided variants. The proofs are equational and mirror Loom's unary-side construction. For `ExceptT` and `OptionT`, the canonical side lifts collapse failure on one side to $\bot$, the lossy "simple framework" choice of Maillard et al. [MHRV20]: it suffices for most cryptographic reductions and avoids the relative-monad infrastructure of their generic framework.

Anchored extension. For proofs needing to reason separately about success and failure (e.g., bounding the probability that a reduction aborts on a non-adversarial event), we provide an *anchored* refinement: two coherence conditions ensuring the relational rwp collapses to the unary wp on the other side when one side is a pure return. These conditions are obligations on the existing rwp (no extra structure) automatically satisfied by the coupling-based `OracleComp` algebras above; they yield *case-aware* variants of the `ExceptT` and `OptionT` side lifts carrying one postcondition per (success/failure) corner of the two-sided computation rather than collapsing failure to $\bot$. This recovers the refined exception semantics of Maillard et al.'s *generic* framework [MHRV20] as derived rules of the existing rwp-algebra. Appendix E carries out the construction.

6.4 Tactics

The structural rules of Definitions 6.1 and 6.2 compose mechanically: each step in a Hoare proof matches a syntactic constructor of the underlying program. Lean 4's standard library now ships such a mechanism, `mvcgen` ("monadic verification-condition generator"), as part of `Std.Do` [The26b]: given a goal $\mathit{pre} \leq \mathsf{wp}\, c\, \mathit{post}$ over `Std.Do.WP` (whose carrier is fixed to Prop), it peels `pure`/`bind`/`if`/`match` by structural induction and looks up user-supplied `@[spec]` lemmas in a discrimination tree to handle non-structural primitives, leaving one side condition per leaf.

`mvcgen`'s carrier is fixed to Prop, so it does not handle quantitative or relational triples. We add a parallel pair of generators, `vcgen` and `rvcgen`, that consume their own `@[vcspec]` spec tables and operate over Definition 6.1 with $L = \mathbb{R}^{\infty}_{\geq 0}$ and over Definition 6.2 respectively. Both share their structural backbone with `mvcgen` but diverge at the leaves: `vcgen` discharges expectation arithmetic; `rvcgen` produces coupling obligations. The relational generator additionally has to *align* two programs whose syntactic shapes need not match: when both sides start with the same query, `rvcstep` couples them with equality; when only one side performs a bind, it falls back to the asynchronous bind rules of Definition 6.2 (the analogues of SSProve's `apply_left`/`apply_right` [HRM+21]); and a user-supplied cut is written `rvcstep using R`. The corresponding qualitative entry points are `by_equiv` (for $\mathsf{evalDist}\, c_1 = \mathsf{evalDist}\, c_2$ goals), `by_dist` (total-variation bounds), and `by_upto` (the identical-until-bad shortcut); the unary entry point is `by_hoare`.

Porting EasyCrypt's tactic vocabulary. The most expressive existing toolset for game-based proofs is EasyCrypt's relational library [BGHZB11], accumulated over more than a decade; its tactics mostly fall into three buckets (structural peelers, semantic shifts, bridges to probability) mapping cleanly onto our generators. Table 3 summarises the correspondence for tactics used in our case studies; each EasyCrypt tactic on the left is a derived rule of our two-monad algebra exposed under the analogous Lean name.

EasyCrypt's tactics operate on a fixed pWHILE syntax with a built-in notion of memory; ours dispatch on the algebraic structure of Definition 6.2 over arbitrary monadic Lean programs, so each entry of Table 3 is justified once and for all rather than re-implemented per language feature. A small example illustrates the resulting proof style: a relational triple between two aligned bind chains closes in three structural `rvcstep`s, one per program constructor, with no manual coupling construction.

```
example (oa : OracleComp spec α)
    (ob : OracleComp spec β) (f : α → β → γ) :
    RelTriple
      (do let x ← oa; let y ← ob; return f x y)
      (do let x ← oa; let y ← ob; return f x y)
      (EqRel γ) := by
  rvcstep -- couple oa with oa
  rvcstep -- couple ob with ob
  rvcstep -- match the pure return on both sides
```

Appendix F works through two longer case-study proofs: the information-theoretic privacy of a two-server PIR protocol via a Boolean negation coupling, and a one-line `mvcgen` discharge of a

EasyCrypt	Role	VCVio
`wp`	peel a deterministic suffix on one side	one-step `rvcstep` (asynchronous bind)
`seq`	cut at a bind with intermediate RHS	`rvcstep using R`
`swap`	reorder independent samplings	`rvcstep` after a permutation lemma
`rnd`	relate two samplings via a bijection	coupling rule for `uniformOfFinset` pairs
`call`	spec-lookup for an external procedure	`@[vcspec]` lemma + `rvcstep`
`byequiv`	probability equality to a coupling	`by_equiv`
`byhoare`	probability bound to a unary triple	`by_hoare`
`transitivity`	multi-step game-hopping	`game_trans`

Table 3: EasyCrypt tactics used in our case studies and their VCVio counterparts.

three-query block of the forking-lemma replay handler. We have not ported every advanced EasyCrypt tactic, partly because the underlying program representations differ. In a foundational setting, this is not a soundness gap, since an unported tactic can always be replaced by an explicit sequence of kernel-checked proof steps.

7 State-Separating Proofs

State-separating proofs (SSP) [BDLF+18] structure a game out of *packages* exposing typed oracle interfaces, so a security argument can swap one component (a PRF, a random oracle, a decisional-assumption oracle) while the rest stays fixed and the proof composes along the same lines. In our framework, packages are recognized as stateful handlers over the same `OracleComp` (Section 7.1); composition consists of two ordinary operations on those handlers (Section 7.2); the reduction lemma is a one-line equality between `OracleComp` programs, not a meta-theorem about advantages; perfect and ε-bounded equivalence between games are lifts of `evalDist` equality and Boolean distinguishing-advantage bounds (Section 7.3); and any relational step inside a package proof is, verbatim, a relational triple from Section 6. Section 7.4 compares with SSProve, with which we share the package-and-link discipline but differ in exactly the four places this design predicts.

7.1 Stateful handlers as packages

A *package* exposes an export oracle interface `E` while internally querying an import interface `I`, maintaining a private state of type σ. We model packages as the unbundled stateful-handler type

```
def QueryHandler.Stateful
    (I E : OracleSpec) (σ : Type) : Type :=
  QueryHandler E (StateT σ (OracleComp I))
```

returning into `StateT σ (OracleComp I)` rather than into a state-passing sub-distribution monad, so the state σ is per-handler and oracle queries are first-class syntax that the program logic can rewrite. This is what lets the same handler stack carry both the package layer and the relational/quantitative triples of Section 6.3: a hop justified by the relational logic transports verbatim to a $\equiv^d$ or $\approx^d_\varepsilon$ hop on packages (Section 7.3). A handler is run against an initial state via `Stateful.run : Stateful I E σ → σ → OracleComp E α → OracleComp I α`, so probabilistic setup such as sampling a long-term key at start-of-game is expressed by passing a sampled state through `run`, with no commitment in the handler's type to whether the state is set up once or per query.

7.2 Composition and the Reduction Lemma

We inherit Brzuska et al.'s two basic compositions [BDLF+18]: sequential $\circ$ runs an outer handler whose imports are an inner handler's exports, and parallel $\mid$ runs two handlers side-by-side under one externally visible interface, taking the disjoint union of their exports and union of their imports (a random oracle queried by both is one oracle, not two [BDLF+18, Def. 7]). What we generalise is the bookkeeping for state and the externally visible interface. Brzuska et al. require the two packages' state-variable sets and output names to be syntactically disjoint, enforced by side conditions and renaming; we replace both by typed data. A *state frame* selects each component's view of the combined state through a non-interfering pair of lenses, recovering the disjoint-state convention as the cartesian-product case. Similarly, an *export route* maps each external query to the component that owns it together with a response-type equivalence, recovering disjoint-output-names as the disjoint-sum case. The downstream API (reduction lemma, equivalence congruences and identical-until-bad bridge of Section 7.3) is stated against the general frame and route; Appendix C.2 gives the formal definitions.

The Fiat-Shamir CMA-to-NMA reduction (Section 8.2.4) exercises both axes at once: the adversary sees one named CMA query type whose constructors enumerate uniform sampling, random-oracle queries, signing, and public-key access, while the underlying components share a single random-oracle cache between adversarial queries and signing-time programming.

The reduction lemma is a program-level identity. The pivotal SSP move is the observation that, for any outer reduction P and inner game Q, running P .link Q against an adversary A is the same program as running Q against the shifted adversary P.shiftLeft A, where `shiftLeft` absorbs P's handler into the adversary:

$$(P \,.\mathsf{link}\; Q).\mathsf{run}\, A \;=\; Q.\mathsf{run}\,(P.\mathsf{shiftLeft}\, A).$$

This is a propositional equality of `OracleComp` programs, not just an `evalDist` equality, and it specialises immediately to the advantage-level statement

$$(P \,.\mathsf{link}\; Q_0).\mathsf{adv}\,(P \,.\mathsf{link}\; Q_1)\, A = Q_0.\mathsf{adv}\, Q_1\,(P.\mathsf{shiftLeft}\, A).$$

A typical SSP-style proof uses this identity to replace a hop on linked games with a hop on the inner games against a shifted adversary, then closes that hop with the program logic of Section 6.

7.3 Equivalence, Bounded Indistinguishability, and Identical-Until-Bad

We expose two equivalence layers between probability-only handlers, keyed on external behaviour rather than internal state shape.

Perfect equivalence $G_0 \equiv^d G_1$ asserts identical output distributions against every adversary on every output type; *ε-indistinguishability* $G_0 \approx^d_\varepsilon G_1$ asserts the Boolean distinguishing advantage is bounded by ε for every Boolean adversary. The state types may differ; what matters is the output distribution. Both relations come with the expected SSP-layer laws (reflexivity, transitivity with ε-budgets summing, monotonicity in ε, congruence under inner-game replacement in `link`, a hybrid lemma collapsing an n-step chain to $\sum_i \varepsilon_i$); Lean's `Trans` machinery lets perfect and bounded hops chain freely in the same proof, with perfect hops contributing 0. A perfect hop is most often discharged by a state bijection plus a per-query `evalDist` equality, the unary specialisation of the relational logic of Section 6.3 to the diagonal coupling. The identical-until-bad bound from Section 6.3 lifts to the package layer as: for two handlers with bad-flagged state $\sigma \times \mathbb{B}$, the distinguishing advantage is bounded by the expected cost of perturbed queries plus the probability of the bad flag firing, with per-step ε allowed to depend on the input state; a constant-ε corollary recovers the textbook $q_S \cdot \varepsilon + \Pr[\mathsf{bad}]$ bound used by the CMA-to-NMA reduction of Section 8.2.4.

7.4 Comparison with SSProve

A preliminary SSP mechanization in F* by Kohbrok et al. [KKRS20] used a library of partial setoids; the discipline was then comprehensively mechanised in Rocq as SSProve [HRM+21, HRVM+23], with recent nominal techniques handling dynamic location allocation [LS25]. Our SSP layer follows SSProve's package-and-link approach but differs in four respects, all downstream consequences of building on `OracleComp` and the program logic of Section 6 rather than on a heap-passing sub-distribution monad.

First, *state separation is structural, not a side condition*: SSProve packages share a single global heap with locations typed via a separate `locs` field, requiring an `fcompat` L_1 L_2 obligation on each composition (and the stronger `fseparate` for the cleanest algebraic laws); recent nominal-SSProve work [LS25] addresses the same friction by quotienting locations modulo alpha equivalence. Our handlers carry their state in their own type, and each external query is owned by exactly one component (by the export route, with the sum tag of `parSum` as the canonical case), so disjointness is forced by typing.

Second, *the reduction lemma is a program-level identity*: SSProve's `Advantage_link` is stated at the advantage level, $\mathsf{Adv}\, G_0\, G_1\, (A \circ P) = \mathsf{Adv}\, (P \circ G_0)\, (P \circ G_1)\, A$; our `run_link_eq_run_shiftLeft` is an equality of underlying `OracleComp` programs, yielding both advantage-level and `evalDist`-level forms as one-line corollaries, while a single application propagates through subsequent measurements without re-proving absorption.

Third, *identical-until-bad is an explicit lemma*: SSProve provides invariant-based RHL machinery (`eq_up_to_inv`, `eq_up_to_inv_adversary_link`) but does not, to our knowledge, have a single named identical-until-bad theorem at the package level with a $q \cdot \varepsilon + \Pr[\mathsf{bad}]$ bound; our bridge is stated once over arbitrary handlers (Section 7.3) and reuses the relational logic for its proof.

Fourth, *handler proofs share infrastructure with the program logic*: every tool from Section 6 (`vcgen`/`rvcgen`, the EasyCrypt-style vocabulary, the coupling rules) applies inside a handler without translation, since handlers and the program logic share the same underlying `OracleComp` syntax; SSProve's RHL is a self-contained semantics that lifts to packages via `eq_up_to_inv`.

8 Case Studies

We illustrate VCVio's oracle manipulation capabilities through three case studies of increasing complexity: a textbook random-oracle commitment scheme that exercises the common handler toolkit (caching, logging, reprogramming); the Bellare–Neven forking lemma, which demonstrates rewinding via deterministic transcript replay; and the Fiat–Shamir transform, structured as a chain of state-separating game hops over the package layer of Section 7, with the Schnorr signature scheme as a corollary.

Threat model. Adversaries are computational: PPT in any asymptotic statements and t-query in the concrete bounds reported below. The standard model is the default; Sections 8.1 and 8.2.4 additionally use the random oracle model, supplied as a named entry in the handler stack via `cachingOracle`. We measure cost in oracle queries; reductions that need finer accounting record an explicit cost transform (Appendix D.2).

Vocabulary. An adversary is an `OracleComp` program, a game runs it against a handler stack to report a win or distinguishing probability, and a reduction is a Lean function on adversaries with a proved advantage inequality. Each case study states security as either a direct quantitative bound on the win probability of a t-query adversary (Section 8.1), or a reduction constructing target adversary B from source adversary A with a proved advantage inequality (Sections 8.2.4 and 8.2.5). A security-parameter-indexed game wrapper, the standard game-hopping and hybrid lemmas, and a polynomial-loss meta-theorem live in Appendix D; none of the case studies depend on this wrapper.

8.1 Random Oracle Commitment Scheme

Consider the standard hash-based commitment scheme in the random oracle model: to commit to a message m, sample a random salt s and output $c = H(m, s)$; to open, reveal (m, s) and check that $H(m, s) = c$. This construction admits clean proofs of binding, extractability, and hiding, each of which exercises a different oracle manipulation technique in VCVio; our formalization follows the textbook treatment of Chiesa and Yogev [CY24, Part IV].

Binding via caching. The binding game gives a t-query adversary access to the random oracle H and asks it to produce a commitment c with two valid openings (m_0, s_0) and (m_1, s_1) where $m_0 \neq m_1$. We model H with `cachingOracle`, which lazily samples and caches a uniform response on each fresh query. Because both adversary queries and the two verification queries run under the same handler, they share a single consistent random function, so a successful break ($(m_0, s_0) \neq (m_1, s_1)$ mapping to the same c) is exactly a collision in the cache. The probability of a collision over $t + 2$ queries into a codomain of size $|C|$ gives $\Pr[\text{binding win}] \leq (t(t-1) + 2)/(2|C|)$.

Extractability via logging. The extractability game splits the adversary into a commit phase producing c and an open phase producing (m, s); the extractor must recover the opening from c and the commit-phase transcript alone. We compose two handlers: `loggingOracle` records query-answer pairs during the commit phase, and `cachingOracle` wraps the whole game so both phases share the same random function. The extractor scans the log for an entry whose answer equals c. The proof splits two cases: a matching entry that disagrees with the adversary's opening is a cache collision (birthday term), while no matching entry means the opening query is fresh and hits c with probability $1/|C|$. The combined bound matches the binding case.

Hiding via reprogramming. The hiding game gives the adversary a message-choosing phase and a distinguishing phase, with a challenge commitment $c = H(m, s)$ in between for a uniform salt s; the adversary wins by distinguishing c from a uniformly random value. The proof uses identical-until-bad with two custom handlers on top of caching, both tracking how many queries have used the challenge salt s: the "real" handler answers normally, while the "simulated" handler redirects every cache miss with salt s to a default query point, decoupling $H(m, s)$ from m without changing the response distribution. The two handlers agree on every query unless the adversary itself queries salt s (the "bad" event, defined as the salt counter reaching 2). Averaged over the uniform choice of s, each of the t queries hits s with probability $1/|S|$, giving $\mathbb{E}_s[d_{\mathrm{TV}}(\mathrm{real}(s), \mathrm{sim}(s))] \le t/|S|$.

8.2 The Forking Lemma

Forking states that an adversary non-negligibly producing a value with some property can be rerun on shared randomness to produce *two* such values that agree up to a designated oracle query, where responses diverge. In Schnorr, two such forgeries on the same message and randomness reveal the secret key.

We mechanize two variants of the Bellare–Neven forking lemma construction [BN06], both exploiting that `OracleComp` represents oracle interactions as an explicit syntax tree, so rewinding reduces to deterministic replay with no need to save or restore adversary state. The seed-based variant `seededFork` (`SeededFork.lean`) follows Bellare and Neven directly by pre-sampling every oracle response into a `QuerySeed` before the first run, and requires the caller to supply a per-oracle query bound (`qb, js`) that the adversary respects. The replay-based variant `forkReplay` (`ReplayFork.lean`) records the first-run transcript live and replays it on the second run with one substituted answer, deriving its bound from the transcript itself rather than from an a priori coverage hypothesis. We use the replay variant for the Fiat–Shamir reduction because it lets the adversary's ambient coin flips share the transcript-based coupling with the managed challenge oracle, without threading an explicit bound through the reduction.

Theorem 8.1 (Forking Lemma, informal). *Let* `main` *be a computation making at most q queries to a designated oracle, and let cf be a function that maps each output to an index $j \le q$ (or aborts). Then there exists an algorithm F that produces two outputs y_1, y_2 such that:*

- *Both are possible outputs of* `main`*;*
- *$cf(y_1) = cf(y_2) = j$ for some j;*
- *The two executions agree on all oracle queries up to and including the j-th, but receive different responses at that query.*

If cf succeeds with probability acc, then F succeeds with probability at least $acc \cdot (acc/q - 1/h)$, with h the size of the oracle's output space.

```
def forkReplay (main : OracleComp spec α)
    (qb : ι → ℕ) (i : ι)
    (cf : α → Option (Fin (qb i + 1))) :
    OptionT (OracleComp spec) (α × α) := do
  let (x₁, log) ← replayFirstRun main
  match cf x₁ with
  | none => failure
  | some s =>
    let u ← $ᵗ spec.Range i
    match log.getQueryValue? i ↑s with
    | none => failure
    | some v =>
        guard (v ≠ u)
        let (x₂, st) ← replayRunWithTraceValue
          main i log ↑s u
        guard (st.forkConsumed ∧ ¬st.mismatch
            ∧ cf x₂ = some s)
        return some (x₁, x₂)
```

Figure 1: The replay-based forking algorithm.

8.2.1 Logging and Replay. The replay variant builds on two oracle handlers that together realize rewinding without any adversary-side state. On the first run, `loggingOracle` wraps the ambient oracle and records every query together with its answer into a `QueryLog`; packaged as `replayFirstRun`, this returns the first-run output paired with the full transcript:

```
def replayFirstRun (main : OracleComp spec α) :
    OracleComp spec (α × QueryLog spec) :=
  (simulateQ loggingOracle main).run
```

On the second run, `replayOracle i` consumes the transcript position by position: if the logged prefix matches it returns the logged answer, except at the fork index (the `s`-th query to oracle family i), where it returns a replacement u. Once the fork is consumed, or if the second run deviates from the prefix, the oracle falls through to the live ambient oracle. The second-run function `replayRunWithTraceValue main i log s u` returns output x_2 and a state recording whether the fork was consumed and whether any mismatch occurred.

8.2.2 The Forking Construction. The forking algorithm (Figure 1) takes a computation `main`, a per-oracle bound `qb`, the oracle family `i` to fork on, and a choice function `cf` that selects a fork index from the output. It runs `main` once via `replayFirstRun` to obtain $(x_1, \mathtt{log})$, samples a fresh oracle response u at the fork index $s = cf(x_1)$, and replays `main` a second time substituting u at position s. The construction aborts whenever any precondition fails: cf does not select a fork, u collides with the logged answer, the transcript has no s-th entry, or the second run deviates from the logged prefix or disagrees with x_1 on the fork index. On success it returns (x_1, x_2).

Because oracle interactions are an explicit syntax tree, the two runs of `main` are deterministically coupled through their shared transcript prefix: `replayOracle` replays the same responses for every query before the fork point, guaranteeing that the computation follows an identical path up to the substituted answer. This is in contrast to the recent EasyCrypt mechanization of Firsov and Janků [FJ25], whose adversary must additionally implement `getState`/`setState` procedures and satisfy an axiomatic predicate `RewProp(A)` declaring that the two together serialize and restore its global state. The pivotal distributional equation underlying their forking proof, that save-run-restore is indistinguishable from the identity, is then taken to "easily follow from the Axioms 1 to 3 of `Rewindable` modules" [FJ25]. In VCVio, the analogous equation is the central lemma we prove *inside* the logic: a distributional coupling showing that two runs of `main` under `replayOracle` with transcripts that agree on their first s entries yield the same distribution on the first s queries and responses, discharged by induction on the syntax tree of the adversary.[5]

8.2.3 Main Result. We prove the Bellare–Neven probability bound for the replay variant under the reachability condition that whenever `cf` selects a fork index, the first-run transcript contains the corresponding query. Writing $acc = \sum_s \Pr[cf(\texttt{main}) = s]$ for the total acceptance probability, $h = |\texttt{spec.Range}\ i|$ for the oracle output space size, and $q = \texttt{qb}\ i + 1$:

```
theorem probOutput_none_forkReplay_le :
    Pr[= none | forkReplay main qb i cf] ≤
      1 - acc * (acc / q - h⁻¹)
```

Equivalently, the success probability is at least $acc \cdot (acc/q - h^{-1})$. The seed-based analogue is proved unconditionally and additionally packaged as a quantitative Hoare triple via the program logic of Section 6. A trace-carrying variant takes the first-run output and transcript as inputs, which the Fiat–Shamir reduction uses to thread the outer query log through the extractor.

The proof follows Bellare and Neven: reduce to a sum over fork indices, bound each term using independence of the re-sampled response, and apply Cauchy–Schwarz. For the replay variant, the per-index bound reduces to two prefix-faithfulness lemmas stating that the log-replayed second run agrees distributionally with the original on the logged prefix; both are proved in `ReplayFork.lean` by induction on the adversary's syntax tree.

8.2.4 Application to Fiat-Shamir. We apply the replay variant to the Fiat–Shamir transform, which turns a Σ-protocol into a signature scheme by deriving the verifier's challenge from a random oracle. A Σ-protocol consists of five operations: the prover's `commit` and `respond` phases, a deterministic `verify`, a simulator `sim` for zero-knowledge, and an `extract`or for special soundness:

```
structure SigmaProtocol
    (Stmt Wit Commit PrvState Chal Resp : Type)
    (rel : Stmt → Wit → Bool) where
  commit (s : Stmt) (w : Wit) :
    ProbComp (Commit × PrvState)
  respond (s : Stmt) (w : Wit)
    (st : PrvState) (c : Chal) : ProbComp Resp
  verify (s : Stmt) (com : Commit)
    (c : Chal) (r : Resp) : Bool
  sim (s : Stmt) : ProbComp Commit
  extract (c₁ : Chal) (r₁ : Resp)
    (c₂ : Chal) (r₂ : Resp) : ProbComp Wit
```

The structure's `sim` field returns only a commitment, used in the completeness specification; the HVZK predicate is parameterized by a separate full-transcript simulator simT (the textbook $\mathsf{Sim}(pk)$ of Appendix G.1 is one instance).

The Fiat–Shamir transform constructs a signature scheme by using a random oracle (modeled as a hash query `M × Commit →ₒ Chal`) to generate the verifier's challenge:

```
def FiatShamir (σ : SigmaProtocol Stmt Wit Commit
      PrvState Chal Resp rel)
    (hr : GenerableRelation Stmt Wit rel) (M : Type)
    [HasQuery (M × Commit →ₒ Chal) m] :
    SignatureAlg m (M := M) (PK := Stmt)
      (SK := Wit) (S := Commit × Resp) where
  keygen := hr.gen
  sign := fun pk sk msg => do
    let (c, e) ← σ.commit pk sk
    let r ← query (msg, c)
    let s ← σ.respond pk sk e r
    return (c, s)
  verify := fun pk msg (c, s) => do
    let r' ← query (msg, c)
    return (σ.verify pk c r' s)
```

In the EUF-CMA game, every query `(msg, c)` above is answered by a single, lazily-sampled random oracle shared between the signing oracle and verification of the forgery, as in the standard random-oracle model. This handler is packaged as `FiatShamir.runtime`, reusing `cachingOracle` from Section 8.1; the advantage `adv.advantage` is taken with respect to it. EUF-CMA is the standard experiment (Appendix G.1), giving the adversary access to ambient and signing oracles.

The reduction wraps the adversary into a managed random-oracle NMA experiment and applies `forkReplay` (via the prefix-faithfulness lemmas of `ReplayFork.lean`) to obtain two forgeries on the same commitment with different challenges; passing them to σ.`extract` recovers a witness, yielding `euf_nma_bound`. A further reduction `euf_cma_to_nma` replaces the signing oracle with the HVZK simulator and routes the verifier's challenge through the live cache, paying $q_S\zeta_{\mathrm{zk}} + q_S(q_S + q_H)\beta$ (Appendix G). Defining $\varepsilon' \coloneqq \varepsilon - q_S\zeta_{\mathrm{zk}} - q_S(q_S + q_H)\beta$, the public theorem `euf_cma_bound` establishes $\varepsilon' \cdot (\varepsilon'/(q_H + 1) - 1/|\mathrm{Chal}|)$ as a lower bound on the probability that extraction succeeds.

8.2.5 The Schnorr Corollary. We instantiate the bound on the textbook Schnorr identification scheme: prime-order group G with generator g, witness space $F = \mathbb{Z}/|G|\mathbb{Z}$, commitment $R = r \cdot g$, challenge $c \in F$, response $z = r + c \cdot sk$, verification $z \cdot g = R + c \cdot pk$. The relation $R(pk, sk) \coloneqq (sk \cdot g = pk)$ is generated by sampling

[5] The concrete EasyCrypt development layers further infrastructure on top of `RewProp`: an imported `RewWithInit.RWI` theory posits a bitstring type `sbits` with two global `axioms` on its pairing operator, `Forking.ec` adds eight more `axioms` for bijective encodings of integers, queries, responses, and state pairs, and an in-file `declare axiom run_prop` posits the distributional property used in the property-transfer step. The adversary must additionally be split by hand into an `init/continue/finish` interface so that `getState` can be inserted between queries; the authors note that the claim that any bounded-query adversary admits such a splitting "cannot be proven in EasyCrypt itself" [FJ25, footnote 7].

Table 4: Case-study formalization size, measured in non-comment Lean lines.

Case study	Defs./Thms.	Proofs	Total
Random-oracle commitments	1,113	3,371	4,484
Forking lemma	860	3,685	4,545
Fiat–Shamir and Schnorr	3,112	5,273	8,385
Total	5,085	12,329	17,414

$sk \leftarrow F$. The three assumptions on the underlying Σ-protocol (special soundness, HVZK, commit-predictability; see Appendix G.2) are discharged with explicit constants: special soundness with the textbook formula $sk = (z_1 - z_2)(c_1 - c_2)^{-1}$; perfect HVZK, so $\zeta_{\text{zk}} = 0$; and commit-predictability $\beta = 1/|F|$. Substituting these into the EUF-CMA bound yields $\varepsilon' = \varepsilon - q_S(q_S + q_H)/|F|$; the theorem `Schnorr.signature_euf_cma` then reduces EUF-CMA security to the discrete-log experiment with bound

$$\varepsilon' \cdot \left(\frac{\varepsilon'}{q_H+1} - \frac{1}{|F|}\right) \le \Pr[\mathrm{DLog}(g, \mathcal{R})].$$

This recovers the textbook Pointcheval–Stern reduction [PS00], mechanized end-to-end; the verbatim Lean statement is in Appendix G.9.

8.3 Remarks on line counts

Table 4 reports formalization size, split into definitions/theorem statements and proofs. The development is larger than the comparable EasyCrypt artifact of Firsov and Janků: for the overlapping forking-lemma and Schnorr material, our Lean formalization is 12,930 non-comment lines vs. their 2,775 (5,721 with the imported rewinding development). Part of this is the foundational price of proving handler semantics, replay invariants, and distributional equalities inside Lean rather than supplying them via axioms or specialized infrastructure. Another factor is AI-assisted authoring, which often closes goals with verbose low-level scripts instead of the shortest idiomatic tactic proof. A natural next step is absorbing routine replay, logging, and counting obligations into the tactic framework, collapsing them to block-level calls as illustrated in Section 6.4 and Appendix F.

9 Related Work

Barbosa et al. [BBB+21] survey computer-aided cryptography, distinguishing between implementation- and design-level verification; we focus on the design level. Within such approaches, *symbolic* tools such as Tamarin [MSCB13] and ProVerif [Bla16] represent messages as atomic terms and primitives as black-box functions, automating adversary reasoning over an equational theory; this is highly automated but bounded by the theory's expressiveness (e.g. a missing equation broke a verified SSH authenticated-encryption scheme [BKN04]). In contrast, *computational* models such as VCVio allow general messages and use programs to express both adversaries and algorithms, yielding stronger real-world guarantees.

Our approach follows other tools embedding protocols in a general-purpose proof assistant (Rocq, Lean, Isabelle) and reasoning within its ambient logic, as in CertiCrypt [ZB11], FCF [Pet15], CryptHOL [BLS19], and SSProve [HRM+21]. This provides the greatest expressivity and modularity, but at a higher proof burden than higher-level tools, since cryptographic reductions are required even when verification can be automated. On the representational side, our `OracleComp` is a structural refinement of interaction trees [XZH+19, yXZH+19], specialized to oracle queries and being inductive rather than coinductive. On the program-logic side, Loom [GPZ+26] introduces *ordered monad algebras* in Lean 4, improving on Maillard et al.'s unary Dijkstra-monad framework [MAA+19]; our *relational ordered monad algebra* matches the expressive power of their relational framework [MHRV20].

In contrast, CryptoVerif [Bla17], EasyCrypt [BGHZB11], Squirrel [BDJ+21], and OWL [GGS+23] provide higher levels of automation while remaining in the computational model, at the cost of a larger unverified codebase and, in some cases, limited modularity. A lighter-weight example is ProofFrog [EMS25], a Python-hosted DSL whose engine canonicalizes game ASTs under a fixed library of semantics-preserving rewrites and dispatches residual obligations to Z3 and SymPy. None of these tools expose an adversary's oracle-interaction history as syntax that can be inspected and replayed, so the forking-lemma-style rewinding argument of Section 8.2 is not directly available, and existing mechanizations (e.g. [FJ25]) pay for the gap with axioms about adversary state.

The Clutch family [GAH+24] develops Iris-based [JKJ+18] program logics for *higher-order* probabilistic programs, spanning asynchronous couplings, error and expected-cost credits [AHdM+24, HLdM+24], apRHL-style approximate reasoning [HLA+25], concurrent extensions [LAG+25, LATB25] (e.g. verifying libsodium's `randombytes_uniform`), and modular sampler specifications [MSBAB26]. The stack treats randomness as a primitive of the operational semantics, so an adversary's randomness-sampling history is not exposed as inspectable, replayable syntax of the kind that powers our forking-lemma argument in Section 8.2. A separate line of probabilistic separation logics [BHL19, LAH23, BDF25] instead interprets the separating conjunction itself as probabilistic independence rather than state separation, and we view their information-theoretic independence and conditional-probability reasoning as highly complementary to our oracle-handler discipline.

Concurrent work. Spitters [Spi26] ports SSProve from Rocq to Lean and claims 172 machine-checked cryptographic security proofs, covering protocols ranging from TLS 1.3 and MLS to NIST post-quantum candidates. CatCrypt's contribution is primarily in breadth and in its Rust-to-Lean extraction pipeline via hax, rather than in framework design, which is inherited directly from SSProve. VCVio's contribution is complementary, focusing on the design of the oracle representation itself. We note also that CatCrypt's repository is not publicly available at the time of writing, so its claims cannot be independently verified. We expect that the shared Lean foundations of both frameworks enables lifting proofs from one system to the other, but this has not yet been verified or implemented. The same oracle API is also used by Dziembowski et al.'s concurrent Lean formalization of computationally sound symbolic cryptography [DFMS25].

10 Conclusion and Future Work

We presented VCVio, a foundational framework for mechanized cryptographic reasoning in Lean. By modeling oracle access as algebraic effects and oracle behavior as handlers, we obtain a flexible toolkit for manipulating computations. To our knowledge, this enables the first foundational mechanization of a general forking lemma and the unforgeability of Schnorr signatures, avoiding the rewindability axioms and explicit state management required by the recent EasyCrypt formalization [FJ25] of Firsov and Janků. VCVio is already being adopted as a base layer for more complex interactive proofs in the ArkLib library [The26a].

For future work, we are expanding our verified primitives to lattice-based encryption (ML-KEM [oST23b]) and signatures (ML-DSA and Falcon [oST23a, FHK+18]), along with a category-theoretic formalization of UC security [Can01]. We also plan to link Lean specifications to efficient executable code, connecting design-level security with implementation-level correctness.

Acknowledgments

We thank Bas Spitters for helpful discussions and feedback on the manuscript. Quang Dao would like to thank LayerZero Labs for supporting his work on this project. This work was supported in part by NSF grant 1814753, an NSF Graduate Research Fellowship, and by a grant from the Etherum Foundation.

References

[AAG05] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing strictly positive types. *Theoretical Computer Science*, 342(1):3–27, 2005.

[ABDG25] Martin Avanzini, Gilles Barthe, Davide Davoli, and Benjamin Grégoire. A quantitative probabilistic relational Hoare logic. *Proc. ACM Program. Lang.*, 9(POPL):1167–1195, January 2025.

[AHdM+24] Alejandro Aguirre, Philipp G. Haselwarter, Markus de Medeiros, Kwing Hei Li, Simon Oddershede Gregersen, Joseph Tassarotti, and Lars Birkedal. Error credits: Resourceful reasoning about error bounds for higher-order probabilistic programs. *Proc. ACM Program. Lang.*, 8(ICFP), August 2024.

[AOBB+25] Santiago Arranz-Olmos, Gilles Barthe, Lionel Blatter, Benjamin Grégoire, Vincent Laporte, and Paolo Torrini. The Jasmin compiler preserves cryptographic security. *CoRR*, abs/2511.11292, 2025.

[AS25] Amir Mohammad Fadaei Ayyam and Michael Sammler. HITrees: Higher-order interaction trees. *CoRR*, abs/2510.14558, 2025.

[BBB+21] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. Sok: Computer-aided cryptography. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 777–795, 2021.

[BCM+26] Clark Barrett, Swarat Chaudhuri, Fabrizio Montesi, Jim Grundy, Pushmeet Kohli, Leonardo de Moura, Alexandre Rademaker, and Sorrachai Yingchareonthawornchai. CSLib: The Lean computer science library, 2026.

[BDF25] Jialu Bao, Emanuele D'Osualdo, and Azadeh Farzan. Bluebell: An alliance of relational lifting and independence for probabilistic reasoning. *Proc. ACM Program. Lang.*, 9(POPL), January 2025.

[BDF+26] Manuel Barbosa, François Dupressoir, Rui Fernandes, Andreas Hülsing, Matthias Meijers, and Pierre-Yves Strub. Completing the chain: Verified implementations of hash-based signatures and their security. Cryptology ePrint Archive, Paper 2026/134, 2026.

[BDJ+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.

[BDLF+18] Chris Brzuska, Antoine Delignat-Lavaud, Cédric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. State separation for code-based game-playing proofs. In *Advances in Cryptology – ASIACRYPT 2018, Part III*, volume 11274 of *Lecture Notes in Computer Science*, pages 222–249. Springer, 2018. Full version: ePrint 2018/306.

[BGHZB11] G. Barthe, B. Gregoire, S. Heraud, and S. Zanella-Beguelin. Computer-aided security proofs for the working cryptographer. *In Proceedings of IACR CRYPTO*, 2011.

[BHL19] Gilles Barthe, Justin Hsu, and Kevin Liao. A probabilistic separation logic. *Proc. ACM Program. Lang.*, 4(POPL), December 2019.

[BKN04] Mihir 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.

[Bla16] B. Blanchet. Modeling and verifying security protocols with the applied pi calculus and proverif. *Foundations and Trends in Privacy and Security*, 2016.

[Bla17] Bruno Blanchet. Cryptoverif: A computationally-sound security protocol verifier. 2017.

[BLS19] David Basin, Andreas Lochbihler, and Reza Sefidgar. Crypthol: Game-based proofs in higher-order logic. 2019.

[BN06] Mihir Bellare and Gregory Neven. New multi-signature schemes and a general forking lemma. In *Proceedings of the 13th Conference on Computer and Communications Security–ACM CCS*, 2006.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145, 2001.

[CY24] Alessandro Chiesa and Eylon Yogev. *Building Cryptographic Proofs from Hash Functions*. 2024.

[DFMS25] Stefan Dziembowski, Grzegorz Fabiański, Daniele Micciancio, and Rafał Stefański. Computationally-sound symbolic cryptography in Lean. Cryptology ePrint Archive, Paper 2025/1700, 2025.

[EMS25] Ross Evans, Matthew McKague, and Douglas Stebila. ProofFrog: A tool for verifying game-hopping proofs. Cryptology ePrint Archive, Paper 2025/418, 2025.

[FHK+18] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, Zhenfei Zhang, et al. Falcon: Fast-Fourier lattice-based compact signatures over NTRU. *Submission to the NIST's post-quantum cryptography standardization process*, 36(5):1–75, 2018.

[FHW25] Simon Foster, Chung-Kil Hur, and Jim Woodcock. Unifying model execution and deductive verification with interaction trees in Isabelle/HOL. *ACM Transactions on Software Engineering and Methodology*, 34(4):1–40, 2025.

[FJ25] Denis Firsov and Jakub Janků. Forking lemma in EasyCrypt. Cryptology ePrint Archive, Paper 2025/573, 2025.

[FU22] Denis Firsov and Dominique Unruh. Reflection, rewinding, and coin-toss in easycrypt. *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2022.

[GAH+24] Simon Oddershede Gregersen, Alejandro Aguirre, Philipp G. Haselwarter, Joseph Tassarotti, and Lars Birkedal. Asynchronous probabilistic couplings in higher-order separation logic. *Proc. ACM Program. Lang.*, 8(POPL), January 2024.

[GGS+23] Joshua Gancher, Sydney Gibson, Pratap Singh, Samvid Dharanikota, and Bryan Parno. Owl: Compositional verification of security protocols via an information-flow type system. *In Proceedings of the IEEE Security and Privacy*, 2023.

[GK13] Nicola Gambino and Joachim Kock. Polynomial functors and polynomial monads. *Mathematical Proceedings of the Cambridge Philosophical Society*, 154(1):153–192, 2013.

[GPZ+26] Vladimir Gladshtein, George Pîrlea, Qiyuan Zhao, Vitaly Kurin, and Ilya Sergey. Foundational multi-modal program verifiers. *Proc. ACM Program. Lang.*, 10(POPL), January 2026.

[HHH+24] Philipp G. Haselwarter, Benjamin Salling Hvass, Lasse Letager Hansen, Théo Winterhalter, Cătălin Hriţcu, and Bas Spitters. The last yard: Foundational end-to-end verification of high-speed cryptography. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2024)*, pages 30–44, 2024.

[HLA+25] Philipp G. Haselwarter, Kwing Hei Li, Alejandro Aguirre, Simon Oddershede Gregersen, Joseph Tassarotti, and Lars Birkedal. Approximate relational reasoning for higher-order probabilistic programs. *Proc. ACM Program. Lang.*, 9(POPL), January 2025.

[HLdM+24] Philipp G. Haselwarter, Kwing Hei Li, Markus de Medeiros, Simon Oddershede Gregersen, Alejandro Aguirre, Joseph Tassarotti, and Lars Birkedal. Tachis: Higher-order separation logic with credits for expected costs. 8(OOPSLA2), October 2024.

[HRM+21] Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Carmine Abate, Nikolaj Sidorenco, Catalin Hritcu, Kenji Maillard, and Bas Spitters. Ssprove: A foundational framework for modular cryptographic proofs in coq. 2021.

[HRVM+23] Philipp G Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Carmine Abate, Nikolaj Sidorenco, Cătălin Hriţcu, Kenji Maillard, and Bas Spitters. SSProve: A foundational framework for modular cryptographic proofs in Coq. *ACM Transactions on Programming*

Languages and Systems, 45(3):1–61, 2023.

[JKJ+18] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018.

[KKRS20] Konrad Kohbrok, Markulf Kohlweiss, Tahina Ramananandro, and Nikhil Swamy. Relational F* for state separating cryptographic proofs. F* wiki article, https://github.com/FStarLang/FStar/wiki/Relational-F*-for-State-Separating-Cryptographic-Proofs, 2020. Accessed 2026-04-26.

[Koc11] Joachim Kock. Polynomial functors and trees. *International Mathematics Research Notices*, 2011(3):609–673, 2011.

[KYW24] Donnacha Oisín Kidney, Zhixuan Yang, and Nicolas Wu. Algebraic effects meet Hoare logic in cubical Agda. *Proceedings of the ACM on Programming Languages*, 8(POPL):1663–1695, 2024.

[LAG+25] Kwing Hei Li, Alejandro Aguirre, Simon Oddershede Gregersen, Philipp G. Haselwarter, Joseph Tassarotti, and Lars Birkedal. Modular reasoning about error bounds for concurrent probabilistic programs. *Proc. ACM Program. Lang.*, 9(ICFP), October 2025.

[LAH23] John M. Li, Amal Ahmed, and Steven Holtzen. Lilac: A modal separation logic for conditional probability. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023.

[LATB25] Kwing Hei Li, Alejandro Aguirre, Joseph Tassarotti, and Lars Birkedal. Contextual refinement of higher-order concurrent probabilistic programs (extended version). arXiv preprint arXiv:2511.10135, 2025.

[LS25] Markus Krabbe Larsen and Carsten Schürmann. Nominal state-separating proofs. In *Proceedings of the 38th IEEE Computer Security Foundations Symposium (CSF 2025)*, pages 363–377, 2025.

[MAA+19] Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Cătălin Hriţcu, Exequiel Rivas, and Éric Tanter. Dijkstra monads for all. *Proc. ACM Program. Lang.*, 3(ICFP), August 2019.

[mC20] The mathlib Community. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 367–381, New York, NY, USA, 2020. Association for Computing Machinery.

[MHRV20] Kenji Maillard, Cătălin Hriţcu, Exequiel Rivas, and Antoine Van Muylder. The next 700 relational program logics. *Proc. ACM Program. Lang.*, 4(POPL), January 2020.

[MSBAB26] Virgil Marionneau, Félix Sassus Bourda, Alejandro Aguirre, and Lars Birkedal. Modular specifications and implementations of random samplers in higher-order separation logic. In *Proceedings of the 15th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP '26, pages 368–382. Association for Computing Machinery, 2026.

[MSCB13] S. Meier, B. Schmidt, C. Cremers, and D. A. Basin. The tamarin prover for the symbolic analysis of security protocols. *In Proc. (CAV)*, 2013.

[NS25] Nelson Niu and David I. Spivak. *Polynomial Functors: A Mathematical Theory of Interaction*, volume 498 of *London Mathematical Society Lecture Note Series*. Cambridge University Press, 2025.

[oST23a] National Institute of Standards and Technology. Module-Lattice-Based Digital Signature Standard. Technical Report Federal Information Processing Standards (FIPS) 204, U.S. Department of Commerce, Washington, D.C., 2023.

[oST23b] National Institute of Standards and Technology. Module-Lattice-Based Key-Encapsulation Mechanism Standard. Technical Report Federal Information Processing Standards (FIPS) 203, U.S. Department of Commerce, Washington, D.C., 2023.

[Pet15] Adam Petcher. A foundational proof framework for cryptography. *Doctoral dissertation, Harvard University, Graduate School of Arts & Sciences*, 2015.

[PP03] Gordon Plotkin and John Power. Algebraic operations and generic effects. *Applied categorical structures*, 11:69–94, 2003.

[PP09] Gordon Plotkin and Matija Pretnar. Handlers of algebraic effects. In *European Symposium on Programming*, pages 80–94. Springer, 2009.

[PS00] David Pointcheval and Jacques Stern. Security arguments for digital signatures and blind signatures. *Journal of cryptology*, 13:361–396, 2000.

[Sch91] Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of cryptology*, 4:161–174, 1991.

[Spi26] Bas Spitters. CatCrypt: From rust to cryptographic security in Lean. Cryptology ePrint Archive, Paper 2026/604, 2026.

[The26a] The ArkLib contributors. Verified proof systems in Lean. Source code repository, 2026. Accessed 2026-04-29.

[The26b] The Lean FRO and contributors. `Std.Do` and `mvcgen`: a Hoare-logic library and verification-condition generator for Lean 4. Source code in the Lean 4 standard library, https://github.com/leanprover/lean4/tree/master/src/Std/Do, 2026. Accessed 2026-04-21.

[XZH+19] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C Pierce, and Steve Zdancewic. Interaction trees: representing recursive and impure programs in coq. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–32, 2019.

[yXZH+19] Li yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees: representing recursive and impure programs in coq. In *Proceedings of the ACM on Programming Languages*, 2019.

[YZZ22] Irene Yoon, Yannick Zakowski, and Steve Zdancewic. Formal reasoning about layered monadic interpreters. *Proceedings of the ACM on Programming Languages*, 6(ICFP):254–282, 2022.

[ZB11] Santiago Zanella-Beguelin. Formal certification of game-based cryptographic proofs. 2011.

[ZNMZ12] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2012)*, pages 427–440, 2012.

A Development Practice and AI-Assisted Formalization

A substantial fraction of the VCVio codebase has been touched by LLM coding agents at some point during development, and external automated proof-search systems have opened pull requests against our repository. This is a methodological fact worth reporting on in its own right, and it also surfaces observations about which parts of a foundational framework's design make it easier or harder for agents to contribute meaningfully.

Setup: workflows and guardrails. The authors use LLM-backed coding environments (primarily Cursor, occasionally Claude Code) in day-to-day development. The repository ships an `AGENTS.md` at its root describing the codebase layout, core invariants, and preferred idioms, pulled in automatically as context by compatible agent harnesses. Every pull request, human- or agent-authored, runs through an automated review pass that flags hygiene issues (unused imports, missing docstrings, style deviations) and, crucially, any `sorry` that has been added or moved; the Lean compiler itself rules out anything that does not type-check.

Where AI helps. We find agents most productive on tasks that are well-scoped and verified by the compiler. Concretely: (i) drafting auxiliary lemmas and routine congruence steps once a human has pinned down the main theorem statement; (ii) meta-programming work on custom tactics and interactive widgets; (iii) literature search, paper synthesis, and drafting documentation prose; and (iv) discharging long mechanical chains such as monad-transformer lifting boilerplate or `simp`-set adjustments across a refactor. Because the compiler rejects anything that does not type-check against the human-authored theorem statement, the correctness cost of an agent-proposed proof is bounded: either it closes the goal or it does not.

External contributions. A non-trivial fraction of our open pull requests originate from automated proof-search systems operated by outside parties, including Harmonic's Aristotle and Logical Intelligence's dispatcher. These systems contribute isolated lemma proofs rather than architectural changes and are accepted subject to the same human review as any other contribution. To the extent that this is a referendum on VCVio's accessibility as a substrate for automated theorem-proving research, the signal is positive.

What does not work. Two failure modes recur. *Unfocused autonomous loops.* Agents given a large, unstructured objective (for instance, "complete the EUF-CMA proof of Schnorr signatures") drift: they produce low-quality intermediate lemmas, spend tokens on irrelevant refactors, or declare success on proofs whose `sorry`s have been pushed deeper into the call graph. Meaningful progress requires the human to decompose the goal into small, locally verifiable sub-objectives and to guard against `sorry`-laundering in review. *Custom tactics are hard to teach.* Our `vcgen` and `rvcgen` tactics (Section 6.4) encode significant domain knowledge, but agents struggle to adopt them idiomatically and tend to fall back on lower-level tactics that solve the immediate goal at the cost of brittle, verbose proofs. This is consistent with the general observation that LLM coding agents are best calibrated against idioms that appear at scale in their pretraining corpus, whereas a newly introduced tactic vocabulary is not.

Implications for framework design. Two observations seem to generalize. First, the free-monad-with-handlers architecture provides a uniform, structural substrate that agents reason over more effectively than ad-hoc stateful encodings: the oracle-handler idiom is regular enough that agents pick up standard patterns from a handful of worked examples. Second, the smallness of a foundational trusted base means that no amount of agent creativity in the proof body can introduce unsoundness; the failure mode of an overconfident agent is wasted effort, not silent bugs. Neither of these is a claim that foundational verification is uniformly AI-native, only that a well-designed foundational framework admits a particular and productive division of labor: humans state theorems, agents propose proofs, the compiler checks.

B Core Lean Definitions

This appendix records the Lean definitions underlying the framework of Sections 3 to 5. We omit universe annotations and some reducibility attributes when they do not affect the mathematical content.

B.1 Polynomial Functors, Free Monads, and Oracle Queries

The implementation represents a set of available oracle queries as a polynomial functor. The head type `A` is the type of query inputs, and `B a` is the response type for query `a`.

```
structure PFunctor where
  A : Type u
  B : A → Type v
```

The free monad over a polynomial functor adds pure leaves to the corresponding well-founded tree type. Internal nodes are positions `a : P.A`, with one subtree for each possible response `u : P.B a`.

```
inductive PFunctor.FreeM (P : PFunctor) :
    Type v → Type _
  | pure {α} (x : α) : FreeM P α
  | roll {α} (a : P.A) (r : P.B a → FreeM P α) :
      FreeM P α
```

The canonical extension into another monad is defined by recursion on this tree. It interprets pure leaves by `pure`, and interprets each internal node by running the supplied handler for its position and continuing with the selected subtree.

```
def PFunctor.FreeM.mapM [Pure m] [Bind m]
    (s : (a : P.A) → m (P.B a)) :
    FreeM P α → m α
  | .pure a => pure a
  | .roll a r => s a >>= fun u => (r u).mapM s
```

An `OracleSpec` is the unbundled version used throughout the library. Its domain is the query-input type, and its range family gives response types.

```
def OracleSpec (ι : Type u) : Type (max u (v + 1)) :=
  ι → Type v

def OracleSpec.toPFunctor (spec : OracleSpec ι) :
    PFunctor :=
  { A := ι, B := spec }

abbrev OracleSpec.Domain (_spec : OracleSpec ι) :
    Type _ := ι
abbrev OracleSpec.Range (spec : OracleSpec ι) (t : ι) :
    Type _ :=
  spec t
```

A single primitive query is an object of the polynomial functor induced by the specification. Equivalently, it is a dependent pair consisting of an input and a continuation consuming the response.

```
def OracleQuery (spec : OracleSpec ι) :
    Type w → Type _ :=
  PFunctor.Obj spec.toPFunctor

def OracleQuery.mk (t : spec.Domain)
    (f : spec.Range t → α) :
    OracleQuery spec α :=
  ⟨t, f⟩

def OracleSpec.query (t : spec.Domain) :
    OracleQuery spec (spec.Range t) :=
  OracleQuery.mk t id
```

The sum of two specifications exposes either set of queries. This is the operation written `spec + spec'` in the main text.

```
instance : HAdd (OracleSpec ι) (OracleSpec ι')
    (OracleSpec (ι ⊕ ι')) where
  hAdd spec spec' := Sum.elim spec spec'
```

B.2 Oracle Computations

The computation monad is the free monad generated by the polynomial functor associated to an `OracleSpec`.

```
def OracleComp (spec : OracleSpec ι) :
    Type w → Type _ :=
  PFunctor.FreeM spec.toPFunctor
```

The implementation exposes the free-monad node constructor as `queryBind`. Thus every computation is either pure or a query followed by a continuation.

```
def OracleComp.queryBind
    (t : spec.Domain)
    (k : spec.Range t → OracleComp spec α) :
    OracleComp spec α :=
  PFunctor.FreeM.roll t k

instance : Monad (OracleComp spec) := inferInstance
instance : LawfulMonad (OracleComp spec) :=
↪ inferInstance
```

The eliminators used in proofs are registered so that induction and case analysis expose exactly the two forms used in Section 3.

```
def OracleComp.recOn
    (oa : OracleComp spec α)
    (pure : (x : α) → motive (pure x))
    (queryBind : (t : spec.Domain) →
      (k : spec.Range t → OracleComp spec α) →
      ((u : spec.Range t) → motive (k u)) →
      motive (OracleComp.queryBind t k)) :
    motive oa
```

B.3 Handlers and Simulation

The Lean formalization names oracle handlers `QueryImpl`. The paper writes `QueryHandler` for the same type to match the algebraic-effects vocabulary.

```
def QueryImpl (spec : OracleSpec ι)
    (m : Type u → Type v) :=
  (t : spec.Domain) → m (spec.Range t)
```

Simulation is the monadic extension of a handler from primitive queries to full computations. It is implemented by the free-monad traversal `PFunctor.FreeM.mapM`.

```
def simulateQ [Monad r]
```

```
    (impl : QueryImpl spec r)
    (mx : OracleComp spec α) : r α :=
  PFunctor.FreeM.mapM impl mx
```

The defining equations are the expected ones for a monad morphism out of a free monad.

```
simulateQ impl (pure x : OracleComp spec α) = pure x

simulateQ impl (mx >>= my) =
  simulateQ impl mx >>= fun x => simulateQ impl (my x)

simulateQ impl (liftM q) =
  q.cont <$> impl q.input
```

B.4 Sub-Specs

A sub-spec embedding is the polynomial-functor lens data used to lift computations from a smaller oracle interface into a larger one. The forward map translates query inputs, and the backward map translates responses.

```
class OracleSpec.SubSpec
    (spec : OracleSpec ι)
    (superSpec : OracleSpec τ)
    extends MonadLift (OracleQuery spec)
        (OracleQuery superSpec) where
  onQuery : spec.Domain → superSpec.Domain
  onResponse : (t : spec.Domain) →
    superSpec.Range (onQuery t) → spec.Range t
  liftM_eq_lift : ∀ (q : OracleQuery spec β),
    monadLift q = ⟨onQuery q.input, q.cont ∘
↪   onResponse q.input⟩
```

The lawful refinement requires each backward map to be bijective. This is the cartesianness condition used in Definition 3.3 and theorem 5.1.

```
class OracleSpec.LawfulSubSpec
    (spec : OracleSpec ι)
    (superSpec : OracleSpec τ)
    [SubSpec spec superSpec] : Prop where
  onResponse_bijective (t : spec.Domain) :
    Function.Bijective (SubSpec.onResponse t)
```

B.5 Distribution Semantics

For arbitrary query distributions, the semantics are another instance of handler simulation.

```
noncomputable def OracleComp.evalDistWhen
    (d : QueryImpl spec SPMF)
    (mx : OracleComp spec α) : SPMF α :=
  simulateQ d mx
```

The default distribution semantics used by `evalDist` sample every oracle response uniformly over its finite inhabited response type.

```
noncomputable instance [spec.Fintype]
↪   [spec.Inhabited] :
    HasEvalPMF (OracleComp spec) where
  toPMF := simulateQ' fun t => PMF.uniformOfFintype
↪   (spec.Range t)
```

```
lemma evalDist_eq_simulateQ (mx : OracleComp spec α) :
    𝒟[mx] =
      simulateQ (fun t => PMF.uniformOfFintype
↪   (spec.Range t)) mx := rfl
```

C Oracle Semantics Details

C.1 Lawful Sub-Specs Preserve Evaluation

Proof of Theorem 5.1. By induction on c. The pure case is immediate from the definition of evalDist. The bind case follows from the induction hypothesis and the bind equation for evalDist.

It remains to consider a single query $t : spec.\mathrm{Domain}$. Lifting the query through the underlying lens ℓ produces $\langle \ell^\sharp\, t,\ k \circ \ell^\flat\, t\rangle$. Evaluating this lifted query pushes the uniform distribution on $spec'.\mathrm{Range}(\ell^\sharp\, t)$ through $\ell^\flat\, t$. Because the sub-spec is lawful, $\ell^\flat\, t$ is a bijection. A bijection between finite types preserves uniformity, so the result coincides with the uniform distribution on $spec.\mathrm{Range}\, t$. □

C.2 General Composition: Frames, Routes, and Ambient Imports

This appendix gives the formal data behind the generalizations to Brzuska et al.'s sequential and parallel composition discussed in Section 7.2. The starting point is the unbundled stateful-handler type $\mathrm{Stateful}\, I\, E\, \sigma \;:=\; \mathrm{QueryHandler}\, E\, (\mathrm{StateT}\, \sigma\, (\mathrm{OracleComp}\, I))$ of Section 7.1. Two pieces of typed data replace the renaming and disjointness side conditions of the original SSP paper [BDLF+18, Definitions 5 and 7]: a *state frame*, which says how each component's private state sits inside the combined state, and an *export route*, which says how each external query name is dispatched to one of the components. Each composition operator below produces another handler of the shape above; the named operators `link`, `parRoute`, `parWithImports`, and `parSum` of the formalization are specializations corresponding to canonical choices for the frame, route, and ambient import interface.

State frames. A *state frame* $F : \mathrm{Frame}\, \sigma\, \sigma_1\, \sigma_2$ is a pair of state lenses $\ell_1 : \sigma \rightleftarrows \sigma_1$ and $\ell_2 : \sigma \rightleftarrows \sigma_2$ (each with a getter and a putter) satisfying the standard very-well-behaved lens laws (put get, get put, put put) together with a *separation* predicate stating that updating one view does not change the other view, and that the two updates commute:

$$\ell_i^\flat\big(\ell_j^\sharp(s, x_j),\, x_i\big) = \ell_j^\sharp\big(\ell_i^\flat(s, x_i),\, x_j\big) \quad \text{for} \quad \{i, j\} = \{1, 2\}.$$

The canonical *product frame* $\mathrm{Frame.prod}\, \sigma_1\, \sigma_2$ takes $\sigma = \sigma_1 \times \sigma_2$ with the projections; a *heap frame* on $\mathrm{Heap}(\alpha \oplus \beta) \simeq \mathrm{Heap}\, \alpha \times \mathrm{Heap}\, \beta$ splits a tagged-key heap into per-tag heaps with the per-cell separation predicate holding definitionally.

Framed sequential composition. Given an outer handler $P : \mathrm{Stateful}\, M\, E\, \sigma_1$, an inner handler $Q : \mathrm{Stateful}\, I\, M\, \sigma_2$, and a frame $F : \mathrm{Frame}\, \sigma\, \sigma_1\, \sigma_2$, the framed link $P\ .\mathrm{linkWith}\ F\ Q$ is a single $\mathrm{Stateful}\, I\, E\, \sigma$ handler. On a query $t : E.\mathrm{Domain}$ and a combined state $s : \sigma$, it reads the outer state $\ell_1^\sharp s$ and the inner state $\ell_2^\sharp s$, runs the nested simulation, and writes the updated component states back through the two lenses:

$$((P\ .\mathsf{linkWith}\ F\ Q)\ t).\mathsf{run}\ s\ =\ F.\mathsf{reshape}\ s\ \big\langle Q.\mathsf{run}\ (\ell_2^\sharp s)\ (P.\mathsf{run}\ (\ell_1^\sharp s)\ (P\ t))\big\rangle.$$

Plain link is the specialization linkWith (Frame.prod $\sigma_1\ \sigma_2$). The structural identity

$$\mathsf{simulateQ}_{\,P.\mathsf{linkWith}\,F\,Q} = \mathsf{simulateQ}_Q \circ \mathsf{simulateQ}_P$$

(modulo F.reshape) is proved by induction on the source program.

Export routes. An *export route* R : ExportRoute $E\,E_1\,E_2$ assigns each external query $t : E.\mathsf{Domain}$ to one component, together with a response-type equivalence:

$$R.\mathsf{route}(t)\ \in\ \big(\Sigma\, t_1 : E_1.\mathsf{Domain}.\ E.\mathsf{Range}\ t \simeq E_1.\mathsf{Range}\ t_1\big) \uplus \big(\Sigma\, t_2 : E_2.\mathsf{Domain}.\ E.\mathsf{Range}\ t \simeq E_2.\mathsf{Range}\ t_2\big).$$

Two stronger variants tighten the route: a *lawful* route additionally requires the underlying tagged-target map $E.\mathsf{Domain} \to E_1.\mathsf{Domain} \uplus E_2.\mathsf{Domain}$ to be injective (no two external names alias one component query, useful for transporting query bounds), and an *equivalence* route requires it to be a bijection (every component query has a unique external name). The canonical *sum route* ExportRoute.sum on $E_1 + E_2$ takes Sum.inl and Sum.inr with identity equivalences.

Routed parallel composition with ambient imports. Given a frame F : Frame $\sigma\,\sigma_1\,\sigma_2$, an export route R : ExportRoute $E\,E_1\,E_2$, an ambient import spec I with embeddings $I_1 \hookrightarrow I$ and $I_2 \hookrightarrow I$, and component handlers h_i : Stateful $I_i\,E_i\,\sigma_i$, the routed parallel composition parRouteWith $F\,R\,h_1\,h_2$: Stateful $I\,E\,\sigma$ dispatches each external query t through R: if $R.\mathsf{route}(t) = \mathsf{inl}\,\langle t_1, \phi\rangle$, it runs $(h_1\ t_1).\mathsf{run}\ (\ell_1^\sharp s)$ under the embedding into I, transports the response back through ϕ^{-1}, and updates only the left view via $\ell_1^\flat$; the right case is symmetric. The component imports are embeddings supplied by MonadLiftT instances; if $I = I_1 + I_2$ and the embeddings are Sum.inl, Sum.inr, the two component oracles remain disjoint, but in general the two components may share inner queries (e.g., a single random oracle).

Specializations recovering named combinators. The named combinators of the formalization arise from particular choices along the three axes:

- link = linkWith (Frame.prod $\sigma_1\ \sigma_2$): canonical product state, sequential composition.
- parRoute R = parRouteWith (Frame.prod $\sigma_1\ \sigma_2$) R: canonical product state, custom route, ambient imports.
- parWithImports F = parRouteWith F ExportRoute.sum: general state, sum-shaped exports, ambient imports.
- parSumWith F = parRouteWith F ExportRoute.sum with $I = I_1 + I_2$ and the canonical sum embeddings: general state, sum exports, sum imports.
- parSum = parSumWith (Frame.prod $\sigma_1\ \sigma_2$): the textbook all-canonical case.

Optional disjointness for sum-like reasoning. Some downstream proofs need the stronger hypothesis that the two ambient embeddings have disjoint query images, even when I itself is not a literal sum. This is exposed by a refinement parRouteSeparatedWith of parRouteWith that takes lawful sub-spec instances $I_1 \subset_o^\ell I$ and $I_2 \subset_o^\ell I$ together with a disjointness hypothesis $I_1 \perp_I I_2$ on their query images; the implementation is identical, and the extra typeclass arguments make sum-like algebraic laws (e.g., the parallel-interchange lemma) available to client developments.

Use of the general API in case studies. The Fiat-Shamir CMA-to-NMA reduction of Section 8.2.4 exercises every axis at once. The combined CMA state is a record (log, cache, keypair, bad), and a frame cmaFrame selects the signed-message log and the inner NMA state through two non-interfering record lenses. The CMA adversary sees a single named query type with constructors unif, ro, sign, pk, routed to the public-forwarding component or the signing simulator by an export route cmaRoute. Both components import the same ambient nmaSpec, so the random-oracle cache that the adversary reads through ro and the cache that the signing simulator programs through the inner programming oracle are the same cache by typing, not by a separate side condition.

D Security Experiments and Games

A SecurityExp is simply an advantage curve indexed by the security parameter. An experiment is *secure* when its advantage is negligible:

```
structure SecurityExp where advantage : ℕ → ℝ≥0∞
def SecurityExp.secure (ase : SecurityExp) : Prop :=
  negligible ase.advantage

structure SecurityGame (Adv : Type*) where
  advantage : Adv → ℕ → ℝ≥0∞
def SecurityGame.secureAgainst (g : SecurityGame Adv)
    (isPPT : Adv → Prop) : Prop :=
  ∀ A, isPPT A → negligible (g.advantage A)
```

The efficiency predicate isPPT is left abstract; users specialize it to polynomial query bounds, polynomial-time cost models, or other notions as appropriate. Smart constructors lift concrete game formulations (failure-based, distinguishing, guessing) into the abstract SecurityGame interface.

D.1 Meta-Theorems for Game-Based Proofs

With this abstraction, the standard meta-theorems for game-based proofs become short, reusable lemmas.

Security reduction. If there is a map reduce : Adv âĘŠ Adv' that preserves efficiency and the advantage of the source game is pointwise bounded by that of the target game on the reduced adversary, then security of the target implies security of the source:

```
theorem secureAgainst_of_reduction
    (hreduce : ∀ A, isPPT A → isPPT' (reduce A))
    (hbound : ∀ A n, g.advantage A n ≤
        g'.advantage (reduce A) n)
    (hsecure : g'.secureAgainst isPPT') :
    g.secureAgainst isPPT
```

Game-hopping. If consecutive games differ by at most a negligible amount $\varepsilon(n)$ in advantage, and the final game is secure, then the first game is also secure:

```
theorem secureAgainst_of_close (hε : negligible ε)
```

```
    (hclose : ∀ A, isPPT A → ∀ n,
      g₁.advantage A n ≤ g₂.advantage A n + ε n)
    (hsecure : g₂.secureAgainst isPPT) :
    g₁.secureAgainst isPPT
```

Hybrid argument. Given a chain of $k+1$ games where consecutive games differ by at most $\varepsilon(n)$, security of the last game implies security of the first. The total advantage loss is at most $k \cdot \varepsilon(n)$, which remains negligible since k is constant:

```
theorem secureAgainst_of_hybrid
    (hε : negligible ε) (k : ℕ)
    (hconsec : ∀ j < k, ∀ A, isPPT A →
      ∀ n, (games j).advantage A n ≤
        (games (j+1)).advantage A n + ε n)
    (hsecure : (games k).secureAgainst isPPT) :
    (games 0).secureAgainst isPPT
```

Polynomial-loss reduction. When the advantage is amplified by a polynomial factor (e.g., from guessing which of $\mathrm{poly}(n)$ hybrid steps to exploit), the closure of negligible functions under polynomial multiplication absorbs the loss:

```
theorem secureAgainst_of_poly_reduction
    (hreduce : ∀ A, isPPT A → isPPT' (reduce A))
    (hbound : ∀ A n, g.advantage A n ≤
        loss.eval n * g'.advantage (reduce A) n)
    (hsecure : g'.secureAgainst isPPT') :
    g.secureAgainst isPPT
```

Each of these theorems is proved in a few lines using the closure properties of negligible functions, but they capture the core reasoning patterns that appear in virtually every game-based security proof.

D.2 Cost-Aware Reductions

A security reduction must not only bound the advantage but also show that the reduced adversary is efficient whenever the original one is. We formalize this with `ReductionWithCost`, which bundles the adversary map with an explicit cost transform:

```
structure ReductionWithCost
    (cost : Adv → ℕ → σ)
    (cost' : Adv' → ℕ → σ') where
  reduce : Adv → Adv'
  transform : ℕ → σ → σ'
  monotone_transform : ∀ n, Monotone (transform n)
  cost_bound : ∀ A n, cost' (reduce A) n ≤
      transform n (cost A n)
```

The type parameters σ and σ' are abstract: they can be scalars, structured resource profiles, or any preordered type. The `transform` field describes how the reduction's overhead translates source cost bounds into target cost bounds, and `cost_bound` certifies that the actual cost of the reduced adversary respects this transform.

Cost-aware reductions compose by composing both the adversary map and the cost transform. The main theorem states that if the cost transform sends the source efficiency class into the target class (captured by `CostClassMap`), then security of the target game for efficient adversaries implies security of the source:

```
theorem secureAgainst_of_reduction_withCost
    (R : ReductionWithCost cost cost')
    (hadv : ∀ A n, g.advantage A n ≤
        g'.advantage (R.reduce A) n)
    (hmap : CostClassMap isEff isEff' R.transform)
    (hsecure : g'.secureAgainst (EfficientFor cost'
↪   isEff')) :
    g.secureAgainst (EfficientFor cost isEff)
```

D.3 Resource Profiles and Query Tracking

To instantiate the abstract cost framework, we provide structured resource profiles that separate intrinsic computation cost from oracle usage:

```
structure ResourceProfile (ω κ : Type*) where
  intrinsic : ω
  usage : κ →₀ ℕ
```

Here `intrinsic` measures the computation's own cost (e.g., runtime), while `usage` is a finitely supported function counting how many times each external capability `k : κ` is invoked. This separation is useful for reduction proofs: we can analyze a reduction body in terms of its intrinsic overhead and interface usage, then instantiate the interface calls with the cost profiles of the subroutines that implement them.

For query counting specifically, a `countingOracle` wraps each oracle call in a `WriterT` layer that accumulates a `QueryCount`. The predicate `IsPerIndexQueryBound oa qb` asserts that computation `oa` makes at most `qb t` queries to each oracle `t`, and its soundness is verified against the counting oracle instrumentation. The asymptotic version `PolyQueries` requires each per-oracle bound to be polynomial in the security parameter, providing the standard notion of "PPT adversary" for oracle-relative settings.

E The Anchored Extension and Case-Aware Exception Reasoning

This appendix gives a self-contained development of the *anchored* extension to the relational ordered monad algebras of Definition 6.2. The extension recovers the case-aware exception semantics of Maillard et al.'s generic framework [MHRV20] without invoking their relative-monad infrastructure.

E.1 Background and motivation

The `ExceptT` and `OptionT` side lifts of Section 6.3 are derived once and for all from the underlying rwp-algebra. They are convenient but *lossy*: failure on one side is identified with the lattice's bottom element $\bot$, so a relational triple between two computations, one of which throws while the other returns successfully, is automatically vacuous. For most cryptographic reductions, this is the right behavior: failures abort the game on both sides, matching what Maillard et al. call the *simple framework*.

There are, however, proofs that need to reason about both branches separately. A typical example is bounding the probability that an honest, non-adversarial check fails: the failure event on one side has nontrivial probability mass that we need to quantify, not collapse to $\bot$. Maillard et al. provide a refined treatment of exceptions in their *generic framework*, but at the cost of introducing a

relative monad over Set to handle the asymmetry between successful and unsuccessful executions. The relative-monad infrastructure is more general than what the cryptographic case studies in this paper require, and breaks the "ordered algebra on a complete lattice" interface that the rest of our development is built around.

The anchored extension below sits in between. It adds two coherence conditions to an existing rwp-algebra and uses them to derive case-aware exception combinators (one postcondition per success/failure × success/failure corner) whose bind laws cleanly chain success cases relationally and route failure cases through unary wp. The construction stays within ordinary lattice-valued algebras, requires no relative monads, and is automatically discharged for the coupling-based `OracleComp` instances of Section 6.3.

E.2 The anchored class

Definition E.1 (Anchored relational algebra). Let M_1, M_2 be monads, L a complete lattice, and assume that we have an ordered monad algebra giving a unary $\mathsf{wp}_i : M_i\,\alpha \to (\alpha \to L) \to L$ on each side $i \in \{1, 2\}$ (Definition 6.1) and a relational ordered monad algebra rwp on (M_1, M_2, L) (Definition 6.2). The triple $(\mathsf{wp}_1, \mathsf{wp}_2, \mathsf{rwp})$ is *anchored* if it satisfies the two coherence conditions

$$\mathsf{rwp}\,(\mathsf{pure}\,a)\,c_2\,\mathit{post} \;=\; \mathsf{wp}_2\,c_2\,(\mathit{post}\,a), \qquad \text{(Left anchor)}$$

$$\mathsf{rwp}\,c_1\,(\mathsf{pure}\,b)\,\mathit{post} \;=\; \mathsf{wp}_1\,c_1\,(\lambda a.\,\mathit{post}\,a\,b), \qquad \text{(Right anchor)}$$

for all a, b, c_1, c_2, and post.

The two conditions say that the relational rwp knows how to "forget" one side once that side is a Dirac. They are not extra structure on top of rwp: the operator is the same; the conditions are obligations the existing operator must satisfy. We call these conditions "anchors" because each says that when one side of rwp degenerates to a Dirac, the relational specification *anchors* to the unary specification on the other side.

Anchoring is independent of the strictness of the bind law.

The coupling-based `OracleComp` algebras of Section 6.3 are anchored but not strict (the optimal coupling for a composite computation can be more precise than the sequential composition of optimal couplings), while a deterministic specification monad is both anchored and strict. We record anchoring in our development as a separate `Anchored` subclass; the `StrictBind` subclass tracks bind-strictness orthogonally.

Proposition E.2 (`OracleComp` is anchored). *Both the qualitative ($L = \mathsf{Prop}$) and quantitative ($L = \mathbb{R}^{\infty}_{\geq 0}$) coupling-based instances of Definition 6.2 on* `OracleComp` *are anchored, with the unary algebras taken to be the corresponding instances of Definition 6.1 on each side.*

Proof. For the qualitative carrier, the unique coupling of a Dirac $\mathsf{evalDist}\,(\mathsf{pure}\,a) = \delta_a$ with any distribution $\mathsf{evalDist}\,c_2$ is the product distribution $(a, \cdot) \mapsto \mathsf{evalDist}\,c_2$. The relational support condition $\forall (x, y) \in \mathsf{support}\,\gamma,\ R\,x\,y$ then unfolds to $\forall y \in \mathsf{support}\,c_2,\ R\,a\,y$, which is exactly the unary $\mathsf{wp}_2\,c_2\,(R\,a)$ in the Prop instance. The right anchor is symmetric.

For the quantitative carrier, the same uniqueness statement says that the infimum over couplings degenerates to a single coupling, and the resulting expectation specializes to $\mathbb{E}_{y \sim \mathsf{evalDist}\,c_2}[\mathit{post}\,a\,y] = \mathsf{wp}_2\,c_2\,(\mathit{post}\,a)$. □

E.3 Case-aware exception combinators

We now use the anchoring conditions to define case-aware analogues of the lossy `ExceptT` and `OptionT` side lifts. The combinators come in three flavors (two-sided, left-only, right-only) and follow a common template: the relational postcondition is replaced by a tuple of postconditions, one per corner, and the relational rwp is applied to the corresponding case-split aggregator.

Definition E.3 (Two-sided case-aware exception WP). Let $(\mathsf{wp}_1, \mathsf{wp}_2, \mathsf{rwp})$ be a relational algebra on (M_1, M_2, L) and let $\varepsilon_1, \varepsilon_2$ be exception types.

For

$$x : \mathsf{ExceptT}\,\varepsilon_1\,M_1\,\alpha, \quad y : \mathsf{ExceptT}\,\varepsilon_2\,M_2\,\beta,$$

and four postconditions

$$\mathit{post}_{\mathsf{OO}} : \alpha \to \beta \to L, \quad \mathit{post}_{\mathsf{EO}} : \varepsilon_1 \to \beta \to L,$$
$$\mathit{post}_{\mathsf{OE}} : \alpha \to \varepsilon_2 \to L, \quad \mathit{post}_{\mathsf{EE}} : \varepsilon_1 \to \varepsilon_2 \to L.$$

we define

$$\begin{aligned}&\mathsf{rwpExc}\,x\,y\,\mathit{post}_{\mathsf{OO}}\,\mathit{post}_{\mathsf{EO}}\\ &\qquad \mathit{post}_{\mathsf{OE}}\,\mathit{post}_{\mathsf{EE}}\\ &\qquad := \mathsf{rwp}\,x.\mathsf{run}\;y.\mathsf{run}\,\mathsf{excPostBoth}_{\overrightarrow{\mathit{post}}},\end{aligned}$$

where the case-split aggregator $\mathsf{excPostBoth}_{\overrightarrow{\mathit{post}}}$ takes a value in $\mathsf{Except}\,\varepsilon_1\,\alpha \times \mathsf{Except}\,\varepsilon_2\,\beta$ and returns the appropriate one of the four postconditions.

The one-sided variants rwpExcLeft and rwpExcRight are analogous but carry only two postconditions (success and failure on the side that has the transformer); the option-typed analogues rwpOpt, rwpOptLeft, rwpOptRight are the same construction with Option in place of Except and one postcondition per (some/none × some/none) corner.

The pure rules of these combinators are immediate consequences of Definition 6.2 alone: when both sides are syntactic pure or throw, the relational rwp collapses to the appropriate corner postcondition. These rules do *not* require anchoring; they are the relational analogues of the four pure rules $\mathsf{rwpExc}\,(\mathsf{pure}\,a)\,(\mathsf{pure}\,b) = \mathit{post}_{\mathsf{OO}}\,a\,b$, $\mathsf{rwpExc}\,(\mathsf{throw}\,e)\,(\mathsf{pure}\,b) = \mathit{post}_{\mathsf{EO}}\,e\,b$, etc.

E.4 Bind laws under anchoring

The payoff of anchoring is the bind law for the case-aware combinators. Without anchoring, the four-corner structure of rwpExc does not chain cleanly: the mixed corners (one side ok, the other error) cannot be related to anything simpler than the underlying rwp. With anchoring, the mixed corners reduce to case-aware *unary* exception WPs on the still-running side, and the bind law specializes accordingly.

To keep the statement readable, write

$$\overrightarrow{\mathit{post}} = (\mathit{post}_{\mathsf{OO}}, \mathit{post}_{\mathsf{EO}}, \mathit{post}_{\mathsf{OE}}, \mathit{post}_{\mathsf{EE}})$$

for the four-tuple of corner postconditions, and abbreviate

$$\mathsf{rwpExc}\,x\,y\,\overrightarrow{\mathit{post}}$$

for the application of rwpExc to its four corner arguments in this canonical order.

Theorem E.4 (Bind law for the case-aware two-sided exception WP). *Assume* $(\mathsf{wp}_1, \mathsf{wp}_2, \mathsf{rwp})$ *is anchored, and let* x :

$\mathsf{ExceptT}\,\varepsilon_1\,M_1\,\alpha$, $y : \mathsf{ExceptT}\,\varepsilon_2\,M_2\,\beta$, $f : \alpha \to \mathsf{ExceptT}\,\varepsilon_1\,M_1\,\gamma$, $g : \beta \to \mathsf{ExceptT}\,\varepsilon_2\,M_2\,\delta$, *and* $\overrightarrow{post}$ *be any tuple of corner postconditions on* (γ, δ). *Define the four corner aggregators*

$$\begin{aligned} K_{\mathsf{OO}}(a,b) &\coloneqq \mathsf{rwpExc}\,(f\,a)\,(g\,b)\,\overrightarrow{post},\\ K_{\mathsf{EO}}(e,b) &\coloneqq \mathsf{wpExc}_2\,(g\,b)\,(post_{\mathsf{EO}}\,e)\,(post_{\mathsf{EE}}\,e),\\ K_{\mathsf{OE}}(a,e) &\coloneqq \mathsf{wpExc}_1\,(f\,a)\,(\lambda c.\,post_{\mathsf{OE}}\,c\,e)\,(\lambda e_1.\,post_{\mathsf{EE}}\,e_1\,e),\\ K_{\mathsf{EE}}(e_1,e_2) &\coloneqq post_{\mathsf{EE}}\,e_1\,e_2, \end{aligned}$$

and write $\overrightarrow{K}$ *for the tuple* $(K_{\mathsf{OO}}, K_{\mathsf{EO}}, K_{\mathsf{OE}}, K_{\mathsf{EE}})$. *Then*

$$\mathsf{rwpExc}\,x\,y\,\overrightarrow{K} \;\le\; \mathsf{rwpExc}\,(x \gg\!= f)\,(y \gg\!= g)\,\overrightarrow{post}.$$

Here wpExc_i *denotes the unary case-aware exception WP on side* i, *derived from* wp_i *by the same case-split construction at the unary level. The four corner aggregators correspond to the four cases:* (ok, ok) *continues relationally with* rwpExc *on* $(f\,a, g\,b)$; (error, ok) *collapses to the unary* wpExc_2 *on* $g\,b$; (ok, error) *collapses symmetrically to* wpExc_1 *on* $f\,a$; *and* (error, error) *is trivial.*

Proof sketch. Unfold rwpExc on both sides. The left-hand side is an instance of rwp on x.run, y.run whose postcondition is a case-split. Apply the relational bind rule of Definition 6.2, then use (Left anchor) and (Right anchor) to rewrite each mixed corner as a unary wpExc call on the still-running side. The full argument is mechanized as `rwpExc_bind_le`. □

The one-sided bind laws follow the same template with one mixed corner instead of two:

Theorem E.5 (Bind law for the case-aware left-only exception WP). *Under anchoring,*

$$\begin{aligned} &\mathsf{rwpExcLeft}\,x\,y\,\big(\lambda a\,b.\,\mathsf{rwpExcLeft}\,(f\,a)\,(g\,b)\,post_{\mathsf{Ok}}\,post_{\mathsf{Err}}\big)\\ &\qquad\big(\lambda e\,b.\,\mathsf{wp}_2\,(g\,b)\,(post_{\mathsf{Err}}\,e)\big)\\ &\le \mathsf{rwpExcLeft}\,(x \gg\!= f)\,(y \gg\!= g)\,post_{\mathsf{Ok}}\,post_{\mathsf{Err}}. \end{aligned}$$

The rwpOpt analogues replace (ok, error) with (some, none), wpExc_i with the unary case-aware option WP wpOpt_i, and the failure case is parameterized by no extra value (since none carries no payload). The proofs are mechanically the same; we omit the precise statements.

E.5 Pure-side reductions

Anchoring also yields a useful family of *reductions*: when one side of a case-aware combinator is a syntactic pure or throw, the four-corner relational WP collapses to a unary case-aware exception WP on the other side. These rules make it manageable to keep a relational stance only over the part of the program that needs it.

Proposition E.6 (Pure-left reductions). *Under anchoring,*

$$\begin{aligned} &\mathsf{rwpExc}\,(\mathsf{pure}\,a)\,y\,post_{\mathsf{OO}}\,post_{\mathsf{EO}}\,post_{\mathsf{OE}}\,post_{\mathsf{EE}}\\ &\qquad = \mathsf{wpExc}_2\,y\,(post_{\mathsf{OO}}\,a)\,(post_{\mathsf{OE}}\,a),\\ &\mathsf{rwpExc}\,(\mathsf{throw}\,e)\,y\,post_{\mathsf{OO}}\,post_{\mathsf{EO}}\,post_{\mathsf{OE}}\,post_{\mathsf{EE}}\\ &\qquad = \mathsf{wpExc}_2\,y\,(post_{\mathsf{EO}}\,e)\,(post_{\mathsf{EE}}\,e). \end{aligned}$$

The right-side analogues hold symmetrically.

The corresponding rwpOpt statements are identical, with wpOpt in place of wpExc.

E.6 Comparison with Maillard et al.

Maillard et al. [MHRV20] consider exceptions in two places. In the simple framework (their §2), they observe that an exception on one side trivializes the relational triple, the same lossy choice as our `ExceptT` side lifts. In the generic framework (their §4–§5), they reintroduce case-aware reasoning by moving from monads to *relative monads* on Set, allowing the relational specification monad to be parameterized by the success/failure shape of the executions.

The anchored extension recovers a substantial fragment of the generic framework's expressivity within the simple framework's algebra. Concretely:

- The four-corner rwpExc plays the role of the generic framework's relational specification monad on the product of two exception monads, but is built from the ordinary rwp via a case-split aggregator, with no relative-monad machinery.
- The bind law of Theorem E.4 matches the shape of the generic framework's four-corner bind on success, with the mixed corners routed through unary case-aware WPs that sit one step closer to standard pre-Maillard verification.
- The pure-side reductions of Theorem E.6 are the relational analogues of the coupling identity "the unique coupling of a Dirac with any distribution is the product distribution."

In effect, the anchored extension trades the generic framework's full categorical generality for a leaner, lattice-valued formulation that suffices for the case-aware exception reasoning needed by the cryptographic case studies of this paper. Where the generic framework requires the user to choose a relational specification monad, our development reuses the existing rwp-algebra and only asks for a separate unary algebra on each side, which the cryptographic instances already provide. The two coherence conditions of Definition E.1 are the precise extra bookkeeping needed to make this reuse work.

E.7 Mechanization status

The anchored class `Anchored` and its derived case-aware combinators (with their bind laws and reductions) are implemented as a generic monad-transformer-style algebra over an arbitrary base relational specification monad. The Prop- and $\mathbb{R}^{\infty}_{\ge 0}$-valued instances on `OracleComp` are discharged from the Dirac-coupling-uniqueness lemma in our coupling theory layer. A worked example exercising case-aware `rwpExc` / `rwpOpt` reductions on `OracleComp` ships with the development. See Appendix **??** for the artifact layout and the precise module locations.

F Worked Tactic Examples

This appendix gives two longer worked examples to complement the toy proof of Section 6.4: a relational `rvcgen`-style proof of information-theoretic privacy for a two-server PIR protocol, and a one-line `mvcgen` discharge of a multi-query block of the forking-lemma replay handler. Both proofs live in the public VCVio development and are checked end-to-end on every CI run; we reference each by file name so the reader can inspect the surrounding definitions.

F.1 Relational example: privacy of a two-server PIR protocol

The protocol fixes a database $a : \mathrm{Fin}\, N \to W$ over an additive group W of characteristic 2 and a target index i_0. The query generator `pirQuery` folds over `List.finRange` N, sampling a uniform Boolean coin per index and updating two accumulators s, s' so that the symmetric difference of the resulting sets is exactly $\{i_0\}$ (`Examples/SimpleTwoServerPIR.lean`, lines 62–69). Information-theoretic privacy says that the marginal distribution of either server's query set is independent of i_0. We focus on the second server view, whose proof exercises the relational tactics most heavily; the first-server case (lines 192–208 of the same file) is a strictly simpler one-coupling instance.

```
theorem pir_private_snd (i₁ i₂ : Fin N) :
    evalDist (Prod.snd <$> pirQuery i₁) =
    evalDist (Prod.snd <$> pirQuery i₂) := by
  simp only [pirQuery]
  by_equiv            -- reduce evalDist equality to a
↪  relational triple
  rvcstep             -- peel the outer `<$>` / map
  rvcstep             -- enter `foldlM` with EqRel as
↪  the inductive invariant
  · rfl               -- initial accumulators are equal
  · intro j acc₁ acc₂ hS
    by_cases h₁ : j = i₁ <;> by_cases h₂ : j = i₂
    · -- j = i₁ = i₂: identical sides, identity
↪  coupling
      subst h₁; subst h₂
      rvcstep using (fun b : Bool => b)
    · -- j = i₁, j ≠ i₂: negation coupling on the coin
      subst h₁
      rvcstep using (fun b : Bool => !b)
    · -- j ≠ i₁, j = i₂: negation coupling
      subst h₂
      rvcstep using (fun b : Bool => !b)
    · -- j ≠ i₁, j ≠ i₂: identity coupling
      rvcstep using (fun b : Bool => b)
    -- residual obligations on EqRel and bijectivity,
↪  omitted
```

The proof in `Examples/SimpleTwoServerPIR.lean`, lines 220–255, is a faithful port of the EasyCrypt PIR proof in `PIR.ec`, modulo the four observations below.

What `by\_equiv` *does.* The boundary tactic `by_equiv` (Table 3) translates the `evalDist` equality $\mathcal{D}[c_1] = \mathcal{D}[c_2]$ into the relational triple $\{\top\}\, c_1 \sim c_2\, \{\mathrm{EqRel}\}$ with the equality post-relation, where the underlying coupling carrier is Prop. Once the goal is a relational triple, all subsequent steps are closed by the relational tactics of Section 6.4.

What `rvcstep` *does on the outer combinators.* The first two `rvcstep` invocations are pure boilerplate: they peel the outer `Functor.map` and enter `List.foldlM` with `EqRel` as the per-index inductive invariant. The infrastructure for these two steps is the `@[vcspec]` discrimination tree that ships with `rvcgen`; the user only needs to supply the loop invariant on the residual goal. Because the residual goal is a per-index, per-accumulator obligation, the entire body of the loop can be discharged independently of the surrounding fold, which is what Section 6.4 flagged as the main payoff of the algebraic-core lifting.

What `rvcstep using $\phi $` *does on a sampling.* The body of the loop samples one fresh Boolean coin on each side. The variant `rvcstep using $\phi $` closes the corresponding sampling pair by exhibiting an explicit bijection $\phi : \mathbb{B} \to \mathbb{B}$ on the coin domain (the EasyCrypt `rnd` tactic; see Table 3). Three of the four cases use the negation $b \mapsto \neg b$ to flip the meaning of the coin on the right-hand side, exploiting the symmetry of the uniform distribution on $\mathbb{B}$; the remaining case uses the identity bijection. The bijection witness (`Function.bijective_id` or `Bool.involutive_not.bijective`) is the only nontrivial mathematical content of the proof: everything else is bookkeeping that the tactic discharges automatically.

What is not *a tactic step.* The branches `simp [...]; split_ifs; ...` after each `rvcstep using $\phi $` are residual arithmetic obligations on the post-relation `EqRel` that have nothing to do with probability: they are pure equality reasoning over Booleans and lists, the kind of obligation that any port of an EasyCrypt proof leaves behind regardless of the host system.

F.2 Unary example: a one-line `mvcgen` discharge

The forking-lemma replay handler `replayOracle` (`VCVio/CryptoFoundations/ReplayFork.lean`) maintains a triple of pre-recorded answers, a forking index, and a partial trace. Three different invariants are useful in different places, so the per-query specifications (`VCVio/CryptoFoundations/ReplayForkStdDo.lean`, lines 75–120) are stated as three separate theorems, `replayOracle_triple_prefix`, `replayOracle_triple_replacement`, and `replayOracle_triple_immutable`. They are deliberately *not* marked `@[spec]`, since a single discrimination-tree key would force `mvcgen` to pick one and silently drop the others.

Once the right per-query spec is supplied, an arbitrary block of replay queries collapses to a single tactic call:

```
example (i t₁ t₂ t₃ : ι) :
    Std.Do.Triple
      (do let _ ← replayOracle i t₁
          let _ ← replayOracle i t₂
          replayOracle i t₃)
      (pre := fun st => ReplayPrefixInvariant i st)
      (post := fun _ st' =>
                ReplayPrefixInvariant i st') := by
  mvcgen [replayOracle_triple_prefix]
```

The same one-liner (`ReplayForkStdDo.lean`, lines 197–218) discharges the analogous statement for `ReplayReplacementInvariant` by switching the spec hint to `replayOracle_triple_replacement`. Section 6.4 flagged `mvcgen`'s discrimination-tree-driven spec lookup as the mechanism that makes such block-level proofs possible without a custom tactic per handler; the example above is the smallest worked instance of that mechanism in the development. The corresponding whole-program corollaries (`ReplayForkStdDo.lean`, lines 122–193) are then derived from the per-query specs via `simulateQ_triple_preserves_invariant`, with no further user input.

G EUF-CMA Security of the Fiat–Shamir Transform

This appendix gives a self-contained, rigorous proof of EUF-CMA security for the Fiat–Shamir transform applied to a Σ-protocol. It covers the full sequence from definitions through intermediate results (including the forking lemma and game-playing infrastructure) to the final bound. Each result is cross-referenced with its mechanized counterpart in VCVio.

G.1 Definitions

Σ-protocols. A *Σ-protocol* for a relation $R \subseteq \mathrm{Stmt} \times \mathrm{Wit}$ with challenge space Ω consists of five probabilistic algorithms $\sigma = (\mathsf{Com}, \mathsf{Resp}, \mathsf{Vfy}, \mathsf{Sim}, \mathsf{Ext})$:

- $\mathsf{Com}(pk, sk) \to (c, e)$: commitment and prover state.
- $\mathsf{Resp}(pk, sk, e, \omega) \to s$: response given challenge ω.
- $\mathsf{Vfy}(pk, c, \omega, s) \to \{0, 1\}$: deterministic verification.
- $\mathsf{Sim}(pk) \to (c, \omega, s)$: HVZK simulator (parameter to the HVZK predicate; the `SigmaProtocol` structure carries only a commit-only simulator for completeness purposes).
- $\mathsf{Ext}(\omega_1, s_1, \omega_2, s_2) \to w$: special-soundness extractor.

The Lean structure is `SigmaProtocol`.

Hard relation and key generation.
A *generable hard relation* (`GenerableRelation`) is a relation R together with a probabilistic generator $\mathsf{Gen} \to (pk, sk)$ satisfying $R(pk, sk) = 1$ with probability 1. The *hard-relation experiment* for a candidate reduction $\mathcal{R} : \mathrm{Stmt} \to \mathrm{ProbComp}(\mathrm{Wit})$ is:

$$\mathsf{HardRelExp}(\mathcal{R}) \ : (pk, sk) \leftarrow \mathsf{Gen};\ w \leftarrow \mathcal{R}(pk);\ \text{output } R(pk, w).$$

Signature scheme.
A *signature scheme* (`SignatureAlg`) consists of three algorithms $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Vfy})$ over a message space $\mathcal{M}$ and a signature space $\mathcal{S}$.

EUF-CMA: existential unforgeability under chosen-message attack.

Let $\mathcal{A}$ be an adversary with adaptive access to a signing oracle $\mathsf{Sign}(sk, \cdot)$ and any ambient oracles $\vec{O}$ (e.g. a random oracle). The EUF-CMA experiment $\mathsf{Exp}^{\text{EUF-CMA}}_{\mathsf{Sig}}(\mathcal{A})$ is:

(1) $(pk, sk) \leftarrow \mathsf{KeyGen};\ L \leftarrow \emptyset$.
(2) $(m^*, \sigma^*) \leftarrow \mathcal{A}^{\mathsf{SignO}(sk, \cdot), \vec{O}}(pk)$, where $\mathsf{SignO}(sk, m)$ appends m to L and returns $\mathsf{Sign}(pk, sk, m)$.
(3) Output 1 iff $\mathsf{Vfy}(pk, m^*, \sigma^*) = 1$ and $m^* \notin L$.

The EUF-CMA advantage of $\mathcal{A}$ is

$$\mathrm{Adv}^{\text{EUF-CMA}}_{\mathsf{Sig}}(\mathcal{A}) \coloneqq \Pr[\mathrm{Exp}^{\text{EUF-CMA}}_{\mathsf{Sig}}(\mathcal{A}) = 1].$$

The EUF-CMA adversary type is formalized as `unforgeableAdv`.

NMA: no-message attack with a managed random oracle.
The Fiat–Shamir reduction targets a stronger-looking but tightly related flavor: the adversary $\mathcal{B}$ has no signing oracle, but its random oracle is *managed*, meaning every query is logged and the resulting cache is exposed to the win predicate. The managed-RO NMA experiment $\mathsf{Exp}^{\text{NMA}}_{\mathsf{Sig}, q_H}(\mathcal{B})$ is:

(1) $(pk, sk) \leftarrow \mathsf{KeyGen};\ C_{\mathrm{RO}} \leftarrow \emptyset$.
(2) $(m^*, (c^*, s^*)) \leftarrow \mathcal{B}^{H(\cdot)}(pk)$, where $H(x)$ samples a fresh uniform ω when $x \notin C_{\mathrm{RO}}$ and otherwise returns $C_{\mathrm{RO}}[x]$, recording (x, ω) in the cache; $\mathcal{B}$ makes at most q_H hash queries.
(3) Output 1 iff $(m^*, c^*) \in C_{\mathrm{RO}}$ and

$$\sigma.\mathsf{Vfy}\big(pk, c^*, C_{\mathrm{RO}}[(m^*, c^*)], s^*\big) = 1.$$

The NMA advantage is $\mathrm{Adv}^{\text{NMA}}_{q_H}(\mathcal{B}) \coloneqq \Pr[\mathrm{Exp}^{\text{NMA}}_{\mathsf{Sig}, q_H}(\mathcal{B}) = 1]$.

The managed-RO NMA adversary type is formalized as `managedRoNmaAdv`; note that it returns the forgery together with the live cache so that $(m^*, c^*) \in C_{\mathrm{RO}}$ is decidable from the adversary's transcript without a separate query log.

In particular, "NMA" stands for *no-message attack* in the standard cryptographic sense: $\mathcal{B}$ never sees a signing oracle. The managed-RO refinement is what makes the NMA game tight enough for the forking lemma to extract a witness without an extra union bound on the hash query log.

The Fiat–Shamir transform.
Given σ and Gen as above, $\mathsf{FS}[\sigma]$ (`FiatShamir`) is the signature scheme over message space $\mathcal{M}$:

- $\mathsf{Sign}(pk, sk, m)$: sample $(c, e) \leftarrow \sigma.\mathsf{Com}(pk, sk)$, query $\omega \leftarrow H(m, c)$, compute $s \leftarrow \sigma.\mathsf{Resp}(pk, sk, e, \omega)$, output (c, s).
- $\mathsf{Vfy}(pk, m, (c, s))$: query $\omega' \leftarrow H(m, c)$, output $\sigma.\mathsf{Vfy}(pk, c, \omega', s)$.

G.2 Quantitative Assumptions

We parameterize the proof by four quantities:

(i) *Special soundness* (`Speciallysound`). For every pk and two accepting transcripts (c, ω_1, s_1), (c, ω_2, s_2) with $\omega_1 \neq \omega_2$: $\Pr[R(pk, w) = 1 \mid w \leftarrow \sigma.\mathsf{Ext}(\omega_1, s_1, \omega_2, s_2)] = 1$.
(ii) *HVZK with gap ζ_{zk}* (`HVZK`). $\Delta\big(\sigma.\mathrm{real}(pk, sk),\ \mathsf{Sim}(pk)\big) \le \zeta_{\mathrm{zk}}$ for all $(pk, sk) \in R$.
(iii) *Commit-predictability β* (`simCommitPredictability`). For every pk and c^*: $\Pr[c = c^* \mid (c, \omega, s) \leftarrow \mathsf{Sim}(pk)] \le \beta$.

Let $\mathcal{A}$ be an EUF-CMA adversary making at most q_S signing queries and q_H hash queries. Write $\varepsilon \coloneqq \mathrm{Adv}^{\text{EUF-CMA}}_{\mathsf{FS}[\sigma]}(\mathcal{A})$.

G.3 Game-Playing Infrastructure

We use two general-purpose lemmas about oracle simulations, both proved by induction on the `OracleComp` syntax tree.

Lemma G.1 (Fundamental lemma of game playing). Lean: `tvDist_simulateQ_le_probEvent_bad`
Let $impl_1$, $impl_2$ be two stateful oracle implementations with state σ, and let $bad : \sigma \to \mathrm{Prop}$ be a monotone predicate (once set, it stays set). If $impl_1$ and $impl_2$ agree on every query from a non-bad state, then for any computation oa and initial state s_0 with $\neg bad(s_0)$:

$$\Delta\big(sim_1,\ sim_2\big) \ \le\ \Pr[bad \mid sim_1],$$

where sim_j denotes the output distribution of oa under $impl_j$ from s_0.

Lemma G.2 (Expected-slack composition). Lean: `ofReal_tvDist_simulateQ_le_expectedQuerySlack_plus_probEvent_output_bad`

Given the same setup as Theorem G.1, let $\varepsilon : \sigma \times Query \to [0,1]$ assign to each non-bad state s and query type q a per-step total-variation slack: $\Delta\big(impl_1(s,q),\ impl_2(s,q)\big) \le \varepsilon(s,q)$ whenever $\neg bad(s)$. For any computation oa:

$$\Delta\big(sim_1,\ sim_2\big) \ \le\ \mathbb{E}_{sim_1}\Big[\textstyle\sum_q \varepsilon(s_q,q)\Big] + \Pr[b = 1 \mid sim_1],$$

where the sum runs over the queries made along an execution. The constant-per-query specialization (with ε a uniform ε_0 on a query-type subset S of cardinality $\le q_S$ and zero on its complement) recovers $q_S \cdot \varepsilon_0 + \Pr[b = 1 \mid sim_1]$.

This refines Theorem G.1 by paying only for "costly" queries. In the Fiat–Shamir proof, signing queries are ε-close (HVZK) while hash queries are identical (both forward to the same RO cache), so the bound uses $q_S \cdot \zeta_{\mathrm{zk}}$ rather than $(q_S + q_H) \cdot \zeta_{\mathrm{zk}}$. The state-dependent form is what lets us discharge HVZK and the programming-collision union bound in a single coupling step.

G.4 The Replay Forking Lemma

We state the replay-based forking lemma. The construction and intuition are described in Section 8.2; here we give the precise quantitative statement and the key proof ingredients.

Setup. Let $main : \mathrm{OracleComp}(spec, \alpha)$ be a computation making queries to oracles indexed by $spec$. Fix a designated oracle family i with finite output space of size $h := |spec.\mathrm{Range}(i)|$, a query bound $q := qb(i) + 1$, and a *choice function* $cf : \alpha \to \mathrm{Option}(\mathsf{Fin}(q))$ selecting a fork index from the output. Write $p_s := \Pr[cf(main) = s]$ and $\mathrm{acc} := \sum_{s=0}^{q-1} p_s$.

Reachability hypothesis.
The replay bound requires *cf-reachability* (`CfReachable`). For every (x, log) in the support of $\mathrm{replayFirstRun}(main)$ with $cf(x) = \mathrm{some}(s)$, the log contains at least $s + 1$ queries to oracle i. This ensures the fork point is always backed by an actual query.

Theorem G.3 (Replay forking bound). Lean: `le_probEvent_isSome_forkReplay`
Under the reachability hypothesis:

$$\mathrm{acc} \cdot \left(\mathrm{acc}/q - h^{-1}\right) \ \le\ \Pr[\mathrm{forkReplay}(main) = \mathrm{some}(\cdot)].$$

Equivalently, $\Pr[\mathrm{forkReplay}(main) = \mathrm{none}] \le 1 - \mathrm{acc} \cdot (\mathrm{acc}/q - h^{-1})$.

Proof outline. The argument follows Bellare–Neven in three layers.

Per-index lower bound. For each s:

$$p_s^2 - p_s/h \ \le\ \Pr\big[cf(x_1) = cf(x_2) = s \ \big|\ \mathsf{forkReplay}\big]. \qquad (1)$$

This decomposes into three sub-lemmas:

(1) *Square bound:* $p_s^2 \le \Pr[\mathsf{noGuard} = z_s]$, where noGuard is the fork without the collision guard ($v = u$ check). This uses the two *prefix-faithfulness* identities: the replay run's prefix distribution matches the first run's.
(2) *Collision bound:* $\Pr[\mathsf{collision} = s] \le p_s/h$, by independence of the re-sampled response $u \leftarrow \Omega$.
(3) *Structural inequality:*

$$\Pr[\mathsf{noGuard}] \le \Pr[\mathsf{fork}] + \Pr[\mathsf{collision}],$$

since the fork discards only the collision cases.

Combining: $p_s^2 - p_s/h \le \Pr[\mathsf{fork}]$ for each s.

Cauchy–Schwarz aggregation. Sum (1) over $s \in \{0, \dots, q-1\}$:

$$\sum_s (p_s^2 - p_s/h) \ \le\ \Pr[\mathsf{fork} = \mathrm{some}(\cdot)].$$

By Cauchy–Schwarz (finite form): $\mathrm{acc}^2 = (\sum_s p_s)^2 \le q \cdot \sum_s p_s^2$, so $\sum_s p_s^2 \ge \mathrm{acc}^2/q$. Thus $\sum_s (p_s^2 - p_s/h) \ge \mathrm{acc}^2/q - \mathrm{acc}/h = \mathrm{acc} \cdot (\mathrm{acc}/q - 1/h)$.

Property transfer. Any property P that holds for every (x, log) in the support of $\mathrm{replayFirstRun}(main)$ also holds for both outputs (x_1, x_2) of a successful fork (with respective logs log_1, log_2), and the logs satisfy

$$log_1.\mathrm{getQueryValue?}(i, s) \ne log_2.\mathrm{getQueryValue?}(i, s),$$

(the distinguished answers at the fork point differ). □

G.5 Main Theorem

Theorem G.4 (EUF-CMA bound). Lean: `euf_cma_bound`
There exists a witness-finding reduction $\mathcal{R}$ such that

$$\varepsilon' \cdot \left(\frac{\varepsilon'}{q_H + 1} - \frac{1}{|\Omega|}\right) \ \le\ \Pr\big[\mathsf{HardRelExp}(\mathcal{R})\big], \qquad (2)$$

where $\varepsilon' = \varepsilon - q_S\,\zeta_{\mathrm{zk}} - q_S(q_S + q_H)\,\beta$.

The proof composes two lemmas: Theorem G.5 (CMA → NMA) and Theorem G.6 (NMA → extraction).

G.6 Step 1: CMA to NMA

Throughout this step we instrument the EUF-CMA experiment with a random-oracle cache $C : \mathcal{M} \times \mathrm{Commit} \rightharpoonup \Omega$ and a signed-message list L. The post-keygen win event

$$E \ :=\ \big(\exists\, \omega.\ C[(m^*, c^*)] = \omega \wedge \sigma.\mathsf{Vfy}(pk, c^*, \omega, s^*) = 1\big) \wedge m^* \notin L$$

captures a forgery whose verifying challenge is read off the cache.

The reduction's NMA adversary $\mathcal{B}$ (the Lean construction `nmaAdvFromCmaWithFinalQuery`) runs $\mathcal{A}$ with the signing oracle replaced by $\mathsf{Sim}(pk)$, threads C as the joint simulator-and-RO cache, and finally issues one explicit live hash query for (m^*, c^*) before returning. The trailing query forces $(m^*, c^*) \in C_{\mathrm{RO}}$ in every execution, so the EUF-CMA verification step is always a cache hit and there is no separate "verifier guesses a fresh challenge" contribution. The wrapped $\mathcal{B}$ issues at most $q_H + 1$ live hash queries (the source's q_H plus the appended verifier-point query), and the forking lemma is invoked at slot parameter q_H, whose $\mathsf{Fin}(q_H + 1)$ candidate-position set accommodates exactly those $q_H + 1$ queries.

Lemma G.5 (CMA to NMA). Lean: `euf_cma_to_nma`
There exists a wrapped NMA adversary $\mathcal{B}$ issuing $\le q_H + 1$ live hash queries such that

$$\varepsilon \ \le\ \mathrm{Adv}^{\mathrm{NMA}}_{q_H}(\mathcal{B}) + q_S\,\zeta_{\mathrm{zk}} + q_S(q_S + q_H)\,\beta,$$

where $\mathrm{Adv}^{\mathrm{NMA}}_{q_H}(\mathcal{B})$ is `Fork.advantage` *at slot parameter q_H: the managed-RO success probability strengthened by the structural condition that the forgery's hash point appears in the live RO log.*

Proof. We exhibit a stateful joint coupling of the real and simulated CMA experiments, threaded through the shared cache state C, and accumulate a total-variation deficit. The deficit decomposes into two independent contributions; the NMA reduction then identifies the simulated game's win event with $\mathrm{Adv}^{\mathrm{NMA}}_{q_H}(\mathcal{B})$.

HVZK simulation cost. We replace the real signing oracle by the HVZK simulator in a single relational triple, threaded through the shared cache state. At each signing query the joint state evolves either by agreement (cache hit, both sides return the same value) or by a costly step (cache miss). On a costly step, assumption (ii) gives a per-step total variation of at most ζ_{zk} between $\sigma.\mathrm{real}(pk, sk)$ and $\mathsf{Sim}(pk)$. Non-signing queries are identical on both sides and contribute zero. Aggregated across the at-most q_S signing queries, the contribution is $q_S\,\zeta_{\mathrm{zk}}$.

Programming-collision cost. At each signing query, the simulated game programs the cache at (m, c) with the simulated challenge ω. A collision occurs when (m, c) is already cached. By assumption (iii), the simulator's commit marginal at any target value has probability at most β, so the probability of colliding with any one of the at-most $q_S + q_H$ prior cache entries is at most $(q_S{+}q_H)\,\beta$ by a union bound. Aggregated across the at-most q_S signing queries, the contribution is $q_S(q_S{+}q_H)\,\beta$.

The HVZK swap and the programming-collision union bound are paid together via Theorem G.2 in its expected-slack form, with the per-query slack functional bounding the local TV plus the local collision probability. This single-coupling treatment is what tightens the bound relative to a chain that would charge HVZK and the bad event separately on each of the real and simulated sides.

NMA bookkeeping. We construct $\mathcal{B}$ by wrapping $\mathcal{A}$ in a handler that forwards live hash queries to the external RO and absorbs simulator-programmed entries into the managed cache, and then appending one explicit live hash query for (m^*, c^*) after $\mathcal{A}$ returns. The trailing query enforces $(m^*, c^*) \in C_{\mathrm{RO}}$ on every execution, and the simulator-programmed cache entries satisfy $C_{\mathrm{RO}} \subseteq C$ together with $m \in L$ for every $(m, c) \in C \setminus C_{\mathrm{RO}}$. For the event E (cache hit at (m^*, c^*), verification passes, and $m^* \notin L$), the cache-overlap invariant identifies $C[(m^*, c^*)]$ with $C_{\mathrm{RO}}[(m^*, c^*)]$ and the trailing query places its position in the live query log of length at most $q_H + 1$, so `Fork.forkPoint` at slot parameter q_H succeeds. This identifies $\Pr[E]$ with $\mathrm{Adv}^{\mathrm{NMA}}_{q_H}(\mathcal{B})$.

Combining.

$$\varepsilon \;\le\; \mathrm{Adv}^{\mathrm{NMA}}_{q_H}(\mathcal{B}) + q_S\,\zeta_{\mathrm{zk}} + q_S(q_S{+}q_H)\,\beta. \qquad \square$$

G.7 Step 2: NMA to Extraction via Forking

Lemma G.6 (NMA to extraction). Lean: `euf_nma_bound`

For any managed-RO NMA adversary $\mathcal{B}$ and fork slot parameter q_H (the size of the $\mathsf{Fin}(q_H + 1)$ candidate-position set; $\mathcal{B}$ may issue up to $q_H + 1$ live hash queries, as the verify-wrapped adversary does), there exists $\mathcal{R}$ such that

$$\mathrm{Adv}^{\mathrm{NMA}}_{q_H}(\mathcal{B}) \cdot \Big(\frac{\mathrm{Adv}^{\mathrm{NMA}}_{q_H}(\mathcal{B})}{q_H{+}1} - \frac{1}{|\Omega|}\Big) \le \Pr\big[\mathsf{HardRelExp}(\mathcal{R})\big].$$

The reduction $\mathcal{R}(pk)$. Run $\mathcal{B}(pk)$ inside forkReplay, forking on the managed challenge oracle at slot parameter q_H. On success, obtain two traces (x_1, x_2). If $x_1.\mathit{target} = x_2.\mathit{target} = (m, c)$ and the caches yield distinct challenges $\omega_1 \neq \omega_2$, output

$$w \leftarrow \sigma.\mathsf{Ext}(\omega_1, s_1, \omega_2, s_2).$$

Otherwise output $w \leftarrow \mathrm{Wit}$ uniformly.

Proof. Write $\mathrm{acc}(pk) \coloneqq \Pr[\text{fork-point succeeds} \mid \mathcal{B}(pk)]$.

(a) Keygen integrals. Both sides are keygen-marginalized:

$$\mathrm{Adv}^{\mathrm{NMA}}_{q_H}(\mathcal{B}) = \sum_{(pk,sk)} \Pr[\mathsf{Gen} = (pk, sk)] \cdot \mathrm{acc}(pk),$$

$$\Pr[\mathsf{HardRelExp}(\mathcal{R})] = \sum_{(pk,sk)} \Pr[\mathsf{Gen} = (pk, sk)] \cdot \Pr[R(pk, w) \mid \mathcal{R}(pk)].$$

(b) Per-key forking bound. For each pk, apply Theorem G.3 to $\mathit{main} = \mathsf{Fork.runTrace}(\mathcal{B}, pk)$ with the fork-point function $cf = \mathsf{forkPoint}_{q_H}$. The reachability hypothesis is verified from the cache-log lockstep invariant: if the fork-point function succeeds at index s, the outer query log has at least $s + 1$ entries. This gives:

$$\mathrm{acc}(pk) \cdot \Big(\frac{\mathrm{acc}(pk)}{q_H{+}1} - \frac{1}{|\Omega|}\Big) \le \Pr\big[\text{fork succeeds with invariant} \;\big|\; pk\big]. \tag{3}$$

(c) Fork support invariant. Every element in the support of $\mathsf{replayFirstRun}(\mathit{main})$ satisfies the following *fork support invariant*: at a valid fork point s, the managed RO cache at the forgery's target (m, c) contains some challenge ω, the outer query log at position s records the same ω, and $\sigma.\mathsf{Vfy}(pk, c, \omega, s_{\mathrm{resp}}) = 1$. This follows from two properties of Fork.runTrace: the cache-log lockstep, which ensures that the s-th outer-log entry matches the cache at the target; and the verified-implies-verify property of the reduction. By the property transfer theorem (Theorem G.3), both fork outputs (x_1, x_2) satisfy this invariant.

(d) Target equality. Both runs share all counted-oracle responses before the fork point (by replay determinism). The internal query log is a deterministic function of oracle responses, so the two logs agree on their first $s{+}1$ entries. Since forkPoint selects the target from the s-th entry, $x_1.\mathit{target} = x_2.\mathit{target}$, giving $c_1 = c_2 =: c$.

(e) Special soundness. The fork gives two accepting transcripts (c, ω_1, s_1) and (c, ω_2, s_2) with $\omega_1 \neq \omega_2$ (from the fork's collision check and the invariant). By assumption (i), $\Pr[R(pk, w) = 1 \mid w \leftarrow \sigma.\mathsf{Ext}(\omega_1, s_1, \omega_2, s_2)] = 1$, so the extraction branch succeeds with probability 1. Composing with (3):

$$\mathrm{acc}(pk) \cdot \Big(\frac{\mathrm{acc}(pk)}{q_H{+}1} - \frac{1}{|\Omega|}\Big) \;\le\; \Pr[R(pk, w) = 1 \mid \mathcal{R}(pk)]. \tag{4}$$

(f) Jensen integration. Since $f(x) = x(x/c - d)$ is convex for $c > 0, d \ge 0$, Jensen's inequality lifts the per-key bound (4) to the

global one:

$$\begin{aligned} f(\mathop{\mathbb{E}}_{\mathrm{Gen}}[\mathrm{acc}]) &\le \mathop{\mathbb{E}}_{\mathrm{Gen}}\left[f(\mathrm{acc})\right] \\ &\le \mathop{\mathbb{E}}_{\mathrm{Gen}}\left[\Pr[R(pk,w) \mid \mathcal{R}(pk)]\right] \\ &= \Pr[\mathrm{HardRelExp}(\mathcal{R})]. \end{aligned}$$

□

G.8 Composing the Two Steps

Proof of Theorem G.4. Theorem G.5 yields $\mathcal{B}$ with

$$\varepsilon' \le \mathrm{Adv}^{\mathrm{NMA}}_{q_H}(\mathcal{B}).$$

Theorem G.6 yields $\mathcal{R}$ with

$$\mathrm{Adv}^{\mathrm{NMA}} \cdot \left(\frac{\mathrm{Adv}^{\mathrm{NMA}}}{q_H+1} - \frac{1}{|\Omega|}\right) \le \Pr[\mathrm{HardRelExp}(\mathcal{R})].$$

Since $f(x) = x(x/c - d)$ is non-decreasing in x for $d \le 1$ (and $1/|\Omega| \le 1$), substituting $\varepsilon' \le \mathrm{Adv}^{\mathrm{NMA}}$ gives (2). □

G.9 Mechanized Statement

Specialized to the textbook Schnorr identification scheme over a prime-order group, with $\zeta_{\mathrm{zk}} = 0$ (perfect HVZK) and $\beta = 1/|F|$ (uniform commitment marginal), the EUF-CMA bound is the Lean theorem `Schnorr.signature_euf_cma`, reproduced below verbatim from `Examples/Schnorr/Signature.lean` (implicit type-class arguments elided):

```
theorem signature_euf_cma (g : G)
    (hg : Function.Bijective (· • g : F → G))
    (M : Type) [DecidableEq M]
    (adv : SignatureAlg.unforgeableAdv
      (signature F G g M))
    (qS qH : ℕ)
    (hQ : ∀ pk, FiatShamir.signHashQueryBound
      (oa := adv.main pk) qS qH) :
    ∃ red : DLogAdversary F G,
      let ε' := adv.advantage
          (FiatShamir.runtime M)
        - qS * (qS + qH) * (card F : ℝ≥0∞)⁻¹
      ε' * (ε' / (qH + 1 : ℝ≥0∞)
          - (card F : ℝ≥0∞)⁻¹) ≤
        Pr[= true | dlogExp g red]
```

Reading guide.

- `signature F G g M` is the Schnorr Fiat–Shamir signature scheme: the generic `FiatShamir` construction of Section 8.2.4 instantiated at the Schnorr Σ-protocol with the discrete-log generable relation.
- `hg` is the standard "F acts simply transitively on G via g" hypothesis (equivalently, g generates G and $|F| = |G|$); it upgrades the simulator's commit marginal to uniform on G, giving the commit-predictability constant $\beta = 1/|F|$.
- `signHashQueryBound (oa := adv.main pk) qS qH` bounds the source adversary's signing-oracle and random-oracle queries by q_S and q_H respectively (cf. the query-counting paragraph in Section 4).
- `FiatShamir.runtime M` is the EUF-CMA runtime stack introduced in Section 8.2.4: `cachingOracle` on the random oracle composed with the uniform-sampling handler for `unifSpec`, started from an empty cache.
- `(card F : ℝ≥0∞)⁻¹` is the rational $1/|F|$ embedded in the extended non-negative reals; `ENNReal` subtraction truncates at zero, so the bound is trivially satisfied whenever the simulation overhead exceeds ε.
- `dlogExp g red` is the textbook discrete-log experiment: sample $sk \leftarrow F$, give $pk = sk \cdot g$ to `red`, accept iff `red` returns sk.

The theorem is closed (no `sorry`). The command

```
#print axioms Schnorr.signature_euf_cma
```

reports exactly the standard kernel axioms

```
[propext, Classical.choice, Quot.sound].
```