Computationally-Sound Symbolic Cryptography in Lean

STEFAN DZIEMBOWSKI*, University of Warsaw, Poland and IDEAS Institute, Poland
GRZEGORZ FABIAŃSKI*, University of Warsaw, Poland
DANIELE MICCIANCIO†, University of California San Diego, USA
RAFAŁ STEFAŃSKI*, University of Warsaw, Poland

We present a formally-verified (in Lean 4) framework for translating *symbolic* cryptographic proofs into the *computationally-sound* ones. Symbolic cryptography is a well-established field that allows reasoning about cryptographic protocols in an abstract way and is relatively easy to verify using proof assistants. Unfortunately, it often lacks a connection to the computational aspects of real-world cryptography. Computationally-sound cryptography, on the other hand, captures this connection much better, but it is often more complex, less accessible, and much harder to verify formally. Several works in the past have provided a bridge between the two, but, to our knowledge, none of them have been implemented in a proof assistant.

We close this gap by formalizing the translation from symbolic to computationally-sound cryptography in Lean 4. Our framework is based on the work of Micciancio (Eurocrypt, 2010) and Li and Micciancio (CSF, 2018), which builds on the idea of using co-induction (instead of induction) for reasoning about an adversary's knowledge in a symbolic setting. Our work encompasses (1) the formalization of the symbolic cryptography framework, (2) the formalization of the computationally sound cryptography framework, and (3) the formalization of the translation between the two. We also provide (4) an extended example of circuit garbling, which is a well-known cryptographic protocol frequently used in secure multi-party computation.

We believe that our work will serve as a foundation for future research in the area of formal verification of cryptographic protocols, as it enables reasoning about cryptographic protocols more abstractly while still providing a formally verified connection to the computational aspects of real-world cryptography.

1 INTRODUCTION

Indistinguishability [21, 22] is a core concept in cryptography, capturing the fact that many important cryptographic primitives (e.g., symmetric and public-key encryption, commitment schemes, zero-knowledge proof systems) should conceal all partial information about their input. In a nutshell, indistinguishability asserts that two objects (such as probability distributions over bitstrings) cannot be told apart with non-negligible probabilistic advantage by any adversary with bounded computational power. Traditionally, such adversaries are modeled as probabilistic polynomial-time (PPT) interactive Turing machines, and the "non-negligible advantage" is defined as a function that decreases faster than any polynomial in the size of the input. Unfortunately, formalized reasoning using proof assistants is notoriously difficult in these settings. For this reason, even several specialized proof assistants targeted at security protocols (like Proverif [14] and Tamarin [30]) adopt an idealized model of cryptography, and analyze security in a symbolic execution model that abstracts away the computational and probabilistic details of concrete implementations. Unfortunately, this provides limited security guarantees, which hold only against adversaries that respect the symbolic abstraction. In practice, security should hold when applications are run in the presence of adversaries (even those not known at protocol design time) that deliberately try to subvert the system, and are not bound by the rules of the symbolic execution model. So, while symbolic analysis can be helpful to check correctness properties, and detect certain types of attacks, it falls short of providing the security guarantees expected by cryptographers.

*Supported (in part) by the European Research Council (ERC) under the European Union's Horizon 2020 innovation program (grant PROCONTRA-885666)
†Supported in part by NSF Award 2411704.

Other formal analysis tools (like EasyCrypt [8, 9] and CryptoVerif [13]) bridge this gap by directly expressing computational security definitions, cryptographic assumptions, and security reductions, in the style of traditional "paper-and-pencil" proofs employed in theoretical cryptography papers, but within a formal, machine-checked environment. A clear advantage of this method is its generality and flexibility. Supporting the definition of fairly general security assumptions and properties, these frameworks allow for the expression and analysis of a wide range of cryptographic applications. However, while the use of a formal framework and mechanized proof verification provides high security assurance, the process of analyzing the security of a protocol remains relatively complex, even harder than manual security proofs.

A third approach, and the focus of this paper, is based on *computationally sound symbolic analysis*, and tries to offer the best of both worlds: simple symbolic proofs, while at the same time offering strong security guarantees against computational adversaries. The approach was pioneered by Abadi and Rogaway [1], who described a simple language of symbolic cryptographic expressions with the remarkable property that when two expressions are symbolically equivalent, their computational interpretations (obtained by implementing and evaluating the expressions using standard algorithms for encryption and other cryptographic primitives) are indistinguishable by any computationally bounded algorithm. Since then, the viability of the approach has been investigated in a number of follow-up papers, including extensions with other cryptographic primitives [31], and various applications to key distribution protocols [32, 33, 35], access control of XML databases [2], password guessing attacks [10], and even the construction of *Garbled Circuits* [27], a general technique for secure two party computation [29, 42]. However, the approach so far has received considerably less attention than general-purpose symbolic and computational security analysis tools, especially when it comes to formalized and automated reasoning.

1.1 Our Contribution

In this paper, we use general-purpose theorem provers (specifically, the Lean 4 proof assistant [16]) to formalize computationally sound symbolic security proofs of complex protocols, demonstrating that these tools are a perfect fit for the task[1]. The use of general-purpose theorem provers has multiple advantages over specialized proof assistants. Firstly, the general-purpose provers enable the use of a rich set of mathematical libraries that are growing at a rapid pace due to the large community of users. Additionally, they may be easier to use for cryptographers without a strong background in formal verification, as multiple tutorials and resources for learning are available. Finally, their popularity means that they are better suited for future enhancements by machine-learning tools (e.g., [41]) whose quality depends on the size of the training sets. In fact, already in our work, we have used the GitHub Copilot tool[2] to assist us in writing types and definitions. (It was much less helpful for writing the proofs).

Our approach is based on the framework proposed in [27] and involves formalizing the following key elements of this work:

(1) the symbolic cryptography language of [27, 31], including the definitions of cryptographic expressions and symbolic indistinguishability.
(2) the definition of computational indistinguishability of distributions[3], and

[1] Our code can be found at https://github.com/ravst/SymbolicCryptographyLean.
[2] See https://github.com/features/copilot.
[3] Since Lean currently misses a library for reasoning about the running time of algorithms, we use an axiomatic approach to model polynomial time. Up to our knowledge, this is handled similarly in several other tools popular tools, such, as, e.g, Easycrypt, which simply assumes that every code written by the user works in polynomial time.

(3) the proof of the soundness theorem of [27], which states that symbolic indistinguishability (defined in point (1) above) implies computational indistinguishability (see point (2)) for cryptographic expressions, as long as the underlying cryptographic primitives are secure.

We exemplify the approach by applying it to the analysis of Garbled Circuits. (The problem was previously considered in [27], but without formalizing the proofs in a proof assistant.) Circuit garbling [29, 42] is a method to securely perform an arbitrary two-party computation of a boolean circuit, with part of the input belonging to one party and part to the other, without revealing any information about one party's input to the other party (besides what is implied by the result of the computation). Formally this is defined in terms of indistinguishability between the actual and simulated execution of the protocol – see Section 3. In this context, our contribution is as follows:

(4) we formalize a proof of security of the garbled circuits in the symbolic model. Thanks to the use of symbolic indistinguishability, which abstracts away from the computational details of cryptographic security primitives and proofs, this proof is very intuitive.

The statements proven in points (3) and (4) allow us to formally conclude that the garbled circuits are secure against any computationally bounded adversary (assuming security of the underlying primitives).

1.2 Related Work

As mentioned above, the most closely related work is [27], which introduced the approach that we formalize in Lean in this paper. That work, in turn, builds on Micciancio [31], who proposed using co-induction to translate symbolic indistinguishability into computational indistinguishability. The authors of [27] partially automated their security proof: for any given circuit C, they could mechanically verify that the output of the garbling procedure $Garble(C)$ is symbolically indistinguishable from the output of the simulator $Sim(y)$, given only $y = C(x)$. This establishes security for that specific circuit C, and they used it to perform random testing on a circuit-by-circuit basis. However, their approach stopped short of a formal proof covering *all* possible circuits. Instead, they provided a pen-and-paper argument (by structural induction on circuits), leaving the formal verification of this general result as an explicit open problem. This problem is solved in this work.

Formalizing cryptographic protocols and security proofs has been an active area of research for many years, with several different approaches and tools being proposed, see, e.g., [8, 9, 13, 14, 30]. We have already provided a comparison between these works and ours. Additionally, there have been recent works that formalized cryptographic protocols and security proofs in Lean. In particular, [40] formalized the framework for reasoning about and manipulating oracle access (which we utilize in our work), the so-called forking lemma, and the Fiat-Shamir transform. Another example is [7], which formalized the soundness of SNARKs [12] in Lean. Finally, [15] formalized differential privacy in Lean. Secure multiparty computation protocols and the Garbled Circuits were formalized in Easycrypt [3–5, 17, 23]. None of these works, however, took our approach of first formalizing symbolic security and then translating it into computational security.

2 SYMBOLIC CRYPTOGRAPHY

As highlighted above, indistinguishability is a cryptographic notion that expresses the inability of an adversary with bounded computational power to distinguish two objects (such as probability distributions over bitstrings) with non-negligible probabilistic advantage. These adversaries are typically modeled as probabilistic polynomial-time (PPT) interactive Turing machines. However, reasoning about such adversaries using proof assistants like Lean is notoriously difficult. Fortunately, a symbolic approach to indistinguishability already exists and offers a compelling alternative. This

approach abstracts away computational complexity, making it significantly more amenable to formal verification and better aligned with the goals of formal methods.

At the core of the symbolic approach are the *cryptographic expressions*, which are formal terms representing cryptographic operations such as encryption. On one hand, these expressions have a concrete *computational semantics*, allowing each expression to be compiled into a probability distribution over bitstrings that can be sampled and transmitted over a network. On the other hand, cryptographic expressions admit a notion of *symbolic indistinguishability* — a formal relation that does not rely on probabilistic polynomial-time adversaries. This relation is supported by a *soundness theorem*, which states that symbolically indistinguishable expressions yield computationally indistinguishable distributions, allowing one to prove indistinguishability properties in a purely symbolic manner.

In this section, we present the definition of cryptographic expressions, their computational semantics, and their symbolic indistinguishability relation. All of those definitions are taken from [27] and formalized in Lean.

2.1 Cryptographic Expressions

Cryptographic expressions is formal language of terms representing some common cryptographic constructions. Before presenting the formal definition, let us go trough a few examples:

(1) **Key variables.** The term `VarK i` represent a variable storing a cryptographic key, where $i \in \mathbb{N}$ is the variable's identifier (or name). Intuitively, each such key variable represents a uniformly random key – see Subsection 2.2 for more details.
(2) **Bit constants.** Term `Bit 0` and `Bit 1` represent bit constants `0` and `1`, respectively
(3) **Encryption.** The constructor `Enc` represents encrypted values. For example `Enc (VarK 0) (Bit 1)` represents the bit `1` encrypted with a random key.
(4) **Pairs.** The terms can be paired using the constructor (`(_, _)`). For example, the expression (`(VarK 0, Enc (VarK 0) (Bit 1))`) represents a pair of a random key and the bit `1` encrypted with that key.
(5) **Reusing Keys.** The keys can be reused. For example the expression (`(Enc (VarK 0) (Bit 1), Enc (VarK 0) (Bit 0))`) represents a pair of two ciphertexts encrypted with the same random key.
(6) **Bit Variables.** The term `VarB i` represents a bit variable, where $i \in \mathbb{N}$ is the variable's name. Similarly to key variables, each bit variable represents a random bit.
(7) **Conditional Swap.** The expression `Perm B` $\mathtt{X}_1$ $\mathtt{X}_2$ represents (`(`$\mathtt{X}_1$`, `$\mathtt{X}_2$`)`) or (`(`$\mathtt{X}_2$`, `$\mathtt{X}_1$`)`) depending on the value the bit `B`.
(8) **Negation.** The bits can be negated using the constructor `Neg`. For example, the expression (`(Neg (VarB 0), VarB 0)`) represents a pair of two bits, where the first one is the negation of the second one.
(9) **Empty expression.** Finally, `Eps` denotes the empty expression. It does not carry any information, but can be useful as a placeholder (especially in the formalized setting).

Formally the expressions are defined by the following grammar:

$$\mathit{Expr} ::= \mathtt{Bit}\ \{0,1\} \mid \mathtt{VarB}\ \mathbb{N} \mid \mathtt{VarK}\ \mathbb{N} \mid (\mathit{Expr},\ \mathit{Expr}) \mid \mathtt{Perm}\ \mathit{Expr}\ \mathit{Expr}\ \mathit{Expr} \mid \mathtt{Neg}\ \mathit{Expr} \mid \mathtt{Eps}$$

Additionally each expression has a *shape* that serves as the expression's type. Here are the possible shapes:

(1) **Bit.** The shape of `Bit b`, `VarB k` and `Neg B` is called `BitS` – it represents a single bit. (Here B denotes an expression of shape `BitS`.)
(2) **Key.** The shape of `VarK k` is `KeyS` – it represents a single key.

```
inductive Shape : Type
| BitS : Shape -- The shape of a single bit. Denoted as 𝔹.
| KeyS : Shape -- The shape of a key. Denoted as 𝕂.
| PairS : Shape → Shape → Shape -- The shape of a pair.
| EncS : Shape → Shape -- The shape of an encrypted value of a given shape.
| EmptyS : Shape -- The empty shape, used to represent an empty expression.

inductive BitExpr : Type
| Bit : Bool → BitExpr -- A constant bit value.
| VarB : Nat → BitExpr -- A bit variable identified by a natural number.
| Not : BitExpr → BitExpr -- Negation of a bit expression.

inductive Expression : Shape → Type
| BitE : BitExpr → Expression 𝔹 -- Bit expression.
| VarK : Nat → Expression 𝕂 -- Key variable identified by a natural number.
| Pair : Expression s₁ → Expression s₂ → Expression (PairS s₁ s₂)
-- ^Pair of expressions. Denoted as ((s₁, s₂)).
| Perm : Expression 𝔹 → Expression s → Expression s → (PairS s s)
--^ Conditional swap.
| Enc : Expression 𝕂 → Expression s → Expression (EncS s)
--^ Encrypt a value with a given key.
| Hidden : Expression 𝕂 → Expression (EncS s)
--^ A hole, that represents a value encrypted with a key unaccessible to the adversary.
| Eps : Expression EmptyS -- Empty expression
```

Fig. 1. Definition of shapes and cryptographic expressions. (See also: `Expression/Defs.lean`.)

(3) **Enc.** If `K` is an expression of shape `KeyS` and `X` and expression of some shape `s`, then the shape of `Enc K X` is `EncS s` – it represents an encrypted value of shape `s`.
(4) **Pair.** If `X` and `Y` expressions of shapes s_1 and s_2, then the shape of `(X, Y)` is `PairS` s_1 s_2 – it represents a pair of expressions of given shapes. Similarly, if `X` and `Y` are expressions of the same shape `s`, and `B` is an expression of shape `BitS`, then the shape of `Perm B X Y` is `PairS s s`.
(5) **Eps.** The shape of `Eps` is `EpsS` – it represents an empty expression.

The formalization of the cryptographic expressions is presented in Figure 1 using the syntax[4] of Lean 4. This syntax is similar to that of several classic proof assistants, such as Coq, and functional programming languages like Haskell [24] and OCaml [26]. For more on Lean 4, see, e.g., [16] or tutorials available online[5]. The definition is split into three parts. First, we define `Shape`, then we define `BitExpr`, which are expressions representing single bits, and finally we define `Expression s`, which is the type of all expressions of an arbitrary shape `s`.

2.2 Computational Semantics

In this section, we equip the cryptographic expressions with a computational semantics. This serves two purposes: The first one is to clarify the intuitive meaning of cryptographic expressions, and the second one is to set out the foundation for the soundness theorem presented in Section 4.

[4]Sometimes the syntax was slightly simplified for clarity, e.g., by removing implicit arguments, namespaces, and some type annotations. For full definitions, please refer to the provided codebase.
[5]See https://lean-lang.org.

```
structure encryptionFunctions (κ : ℕ) where
  encLength : ℕ -> ℕ
  encrypt : (key : BitVector κ) -> (msg : BitVector n) -> PMF (BitVector (encLength n))
  decrypt : (key : BitVector κ) -> (msg : BitVector (encLength n)) -> BitVector n
```

Fig. 2. Definition of the encryption scheme. (`Expression/ComputationalSemantics/encryption-IndCpa.lean`)

To evaluate a cryptographic expression, we need to select an encryption protocol which is modelled formally in Figure 2. In the listing κ denotes the length of the cryptographic keys (i.e. the security parameter). `BitVector n` denotes a bit vector of a given length, and `PMF X` denotes a distribution (given by a probability mass function) over `X`, so a function of type `X → PMF Y` can be seen as a probabilistic function from `X` to `Y`. In particular `encrypt` is a probabilistic function that takes a key of length κ a message of an arbitrary length n, and returns a ciphertext length of length `encLength n`, where `encLength` is a parameter of the encryption scheme that ensures that the length of the ciphertext does not depend on the content of the message, but only on its length. Figure 2 also defines the decryption function, but it will not be used in the computational semantics.

Once we have fixed the encryption scheme, and we are given the variable valuations (i.e., functions `kVars : ℕ → BitVector k` and `bVars : ℕ → Bool`), we can define the computational semantics of cryptographic expressions as in Figure 3. In this listing, notation `let x ← X` denotes sampling x from a distribution X (more precisely, it is a monadic bind operation on the type `PMF`). We compute the final probability distribution by independently sampling the value of each variable uniformly from `Bool` or `BitVector` κ and then evaluating the expression, as defined in Figure 4. Here `Fin k` denotes the finite set of $\{0, \ldots, k-1\}$. Let us finish this section with a few examples of cryptographic expressions and their computational semantics:

(1) The expression `((Bit 0, Bit 1))` results in the bitstring `01`, with probability 1.
(2) The expression `((VarB 0, VarB 1))` results in the uniform distribution over the bitstrings `00`, `01`, `10`, and `11`, each with probability 1/4. This is because the two elements of the pair are independent random bits.
(3) The expression `((VarB 0, VarB 0))` results in the uniform distribution over the bitstrings `00` and `11`, each with probability 1/2. This is because both the first and the second coordinate of the pair are the same random value.
(4) The expression `Perm (VarB 0) (Bit 1) (Bit 0)` results in the uniform distribution over the bitstrings `10` and `01`, each with probability 1/2. This is because `0` and `1` are swapped according to the random value of `VarB 0`.

2.3 Symbolic Indistinguishability

The final piece of the puzzle is the *symbolic indistinguishability relation*. On one hand, it is a formal way of capturing the intuitive notion of indistinguishability of cryptographic expressions. On the other hand, it can be seen as a sufficient condition for the computational indistinguishability of the resulting probability distributions (provided that the encryption scheme used to evaluate the expressions is secure). In this part of the paper, we will focus on the intuitive meaning of the symbolic indistinguishability relation, and we will focus on its formal guarantees in Section 4.

2.3.1 Examples. Before presenting the formal definition of symbolic indistinguishability (in Section 2.3.2), we begin with a few illustrative examples. Consider first the following pair of expressions:

`Enc (VarK 0) (Bit 0)` and `Enc (VarK 0) (Bit 1)`

```
def evalExpr (enc : encryptionFunctions κ) (kVars : ℕ → BitVector κ)
    (bVars : ℕ → Bool) (e : Expression s) : PMF (BitVector (shapeLength κ enc s)) :=
  match e with
  | Expression.Enc (Expression.VarK k) e => do
    -- Encrypt e using k
    let e' ← evalExpr enc kVars bVars e
    let key := kVars k
    enc.encrypt key e'
  | Expression.Pair e₁ e₂ => do
    -- Concatenate e₁ and e₂
    let e₁' ← evalExpr enc kVars bVars e₁
    let e₂' ← evalExpr enc kVars bVars e₂
    return (List.Vector.append e₁' e₂')
  | Expression.BitE b => do
    -- Lift a bit to a vector
    let b' := evalBitExpr bVars b
    return (List.Vector.cons b' List.Vector.nil)
  | Expression.VarK k => do
    -- Read the value of a key variable
    return (kVars k)
  | Expression.Perm (Expression.BitE b) e₁ e₂ => do
    -- Conditional swap:
    -- Return (e₁, e₂) if b is equal to 1 or (e₂, e₁) otherwise
    let b' := evalBitExpr bVars b
    let e₁' ← evalExpr enc kVars bVars e₁
    let e₂' ← evalExpr enc kVars bVars e₂
    if b' then return (List.Vector.append e₂' e₁')
    else return (List.Vector.append e₁' e₂')
  | Expression.Eps =>
    -- Empty bitstring
    return (List.Vector.nil)
  | Expression.Hidden (Expression.VarK k) => do
    -- Encode an arbitrary vector (e.g. only ones) using the key k.
    -- The idea is that since k is unknown to the adversary,
    -- it does not matter what is encrypted
    let key := kVars k
    enc.encrypt key ones
```

Fig. 3. Definition of computational semantics, given the encryption scheme and variable valuations. (`Expression/ComputationalSemantics/Def.lean`.)

```
def exprToDistr (enc : encryptionFunctions κ) (e : Expression s) : PMF (BitVector
    (shapeLength κ enc s)) := do
  let v := getMaxVar e + 1
  let bvars ← uniformOfFintype (Fin v -> Bool)
  let kvars ← uniformOfFintype (Fin v -> BitVector κ)
  evalExpr enc (extendFin ones kvars) (extendFin false bvars) e
```

Fig. 4. Full definition of the computational semantics of cryptographic expressions with random variable valuations.

Those two expressions should be indistinguishable, because the ciphertexts encrypted with a random key should not leak any information about the plaintext. Next, let us consider a similar situation, but this time the encryption key is published:

`((VarK 0, Enc (VarK 0) (Bit 0)))` and `((VarK 0, Enc (VarK 0) (Bit 1)))`

Those two expressions should count as distinguishable, because the adversary can decrypt the second coordinate using the key from the first coordinate, and check if it is `0` or `1`. As a side note, observe that the two examples show that indistinguishability is not a congruence – even if an expression Y is symbolically indistinguishable from Y', it does not mean that `((X, Y))` is symbolically indistinguishable from `((X, Y'))`.

The encryption can be chained. For example, the following pair of expressions is distinguishable:

`((VarK 0, ((Enc (VarK 0) (VarK 1), Enc (VarK 1) (Bit 0)))))` and
`((VarK 0, ((Enc (VarK 0) (VarK 1), Enc (VarK 1) (Bit 1)))))`

But if we remove the first coordinate from each of the expressions, they become indistinguishable.

As our next example, consider the following pair of expressions:

`((Enc (VarK 0) (Bit 0), Enc (VarK 0) (Bit 1)))` and
`((Enc (VarK 0) (Bit 0), Enc (VarK 0) (Bit 0)))`

Those two expressions should be indistinguishable. This is possible because our encryption function is probabilistic, so even though the first and second coordinates of the second expression are the same, they might (and probably will) result in different ciphertexts.

Let us now focus on a slightly counterintuitive aspect of encryption security: As it turns out, the usual notion of encryption security (i.e. the IND-CPA definition, see Section 4.1) does not protect the security of messages encrypted cyclically. For example, we should assume that encrypting a key with itself (i.e. `Enc (VarK 0) (VarK 0)`) leaks all information about the key, so the following two expressions should be distinguishable:

`((Enc (VarK 0) (VarK 0), Enc (VarK 0) (Bit 0)))` and
`((Enc (VarK 0) (VarK 0), Enc (VarK 0) (Bit 1)))`

The cycle might be more complex than just encrypting a key with itself. Another example is:

`((Enc (VarK 0) (VarK 1), Enc (VarK 1) (VarK 0)))`

And we should assume that this expression leaks the keys `VarK 0` and `VarK 1`.

Finally, let us discuss a few examples that have to do with bits and the conditional swap operations. First, we consider the following pair:

`VarB 0` and `Neg (VarB 0)`

Those two expressions are indistinguishable, because both of them result in the uniform distribution over `0` and `1`. Next, consider the following pair:

`Perm (VarB 0) (Bit 0) (Bit 1)` and `Perm (VarB 0) (Bit 1) (Bit 0)`

Those two expressions are also indistinguishable, because they both result in the uniform distribution over the bitstrings `01` and `10` (each with probability 1/2). If we publish the value of `VarB 0` in the first coordinate, we get the following pair:

`((VarB 0, Perm (VarB 0) (Bit 0) (Bit 1)))` and `((VarB 0, Perm (VarB 0) (Bit 1) (Bit 0)))`

Those two expressions are not indistinguishable, because the adversary can use the first coordinate to reverse the conditional swap of the second coordinate and check if it is `01` or `10`.

2.3.2 Definition. We are now ready to present the formal definition of symbolic indistinguishability. It consists of three parts:

```
def hideEncrypted (keys : Finset (Expression KeyS)) (e : Expression s) : Expression s
    :=
  match e with
  | Pair e1 e2 => Pair (hideEncrypted keys e1) (hideEncrypted keys e2)
  | Perm b e1 e2 => Perm b (hideEncrypted keys e1) (hideEncrypted keys e2)
  | Enc k e =>
    if k ∈ keys
    then Enc k (hideEncrypted keys e)
    else Hidden k
  | e => e -- In all other cases, we leave the expression as is.

def extractKeys (e : Expression s) : Finset (Expression KeyS) :=
  match e with
  | VarK e => {VarK e}
  | Expression.Pair p1 p2 => (extractKeys p1) ∪ (extractKeys p2)
  | Expression.Perm _ p1 p2 => (extractKeys p1) ∪ (extractKeys p2)
  | Expression.Enc _ e => (extractKeys e) -- We omit the key used for encryption
  | _ => ∅ -- In all other cases, we return the empty set.

def keyRecovery (e : Expression s) (S : Finset (Expression KeyS)) : Finset (Expression
    KeyS) :=
  extractKeys (hideEncrypted S e)

def adversaryKeys (e : Expression s) : Finset (Expression KeyS) :=
  -- `greatestFixpoint f X Y Z` returns the greatest fixpoint of `f`,
  -- the other arguments are only needed to prove termination, and do not influence
    the output.
  greatestFixpoint (keyRecovery e) (reductionToOracle e) (keyRecoveryMonotone e)
    (keyRecoveryContained e)

def adversaryView (e : Expression s) : Expression s :=
  hideEncrypted (adversaryKeys e) e
```

Fig. 5. Definition of `adversaryView`. (`Expression/SymbolicIndistinguishability.lean`.)

(1) `adversaryView` function, which hides all parts of the expressions that are not accessible to the adversary, i.e. are encrypted with a key that is not known to the adversary.
(2) `applyVarRenaming` function, which allows us to rename both key and bit variables in the expressions.
(3) `normalizeExpr` function, which performs all possible simplifications of the bit expressions.

Adversary View. As shown in the examples, to hide the inaccessible parts of the expressions, `adversaryView` (defined in Figure 5) needs to handle chained and circular encryption. For this reason, `adversaryView` is defined in terms of the *greatest* fixpoint of an auxiliary `keyRecovery` function (i.e., the $\mathcal{F}_e$ operator from [27]), which in turn is a composition of `hideEncrypted` and `extractKeys`. The function `hideEncrypted` (analog of the function **p** from [27]) inputs an expression e and a set of keys K and hides all occurrences of `Enc k e` where k is not in K. The second function `extractKeys` extracts all the keys that appear in the body of an expression (either as plaintext or encrypted). Importantly `extractKeys` does not extract keys that are only used to encrypt data and do not appear anywhere else in the expression. For example:

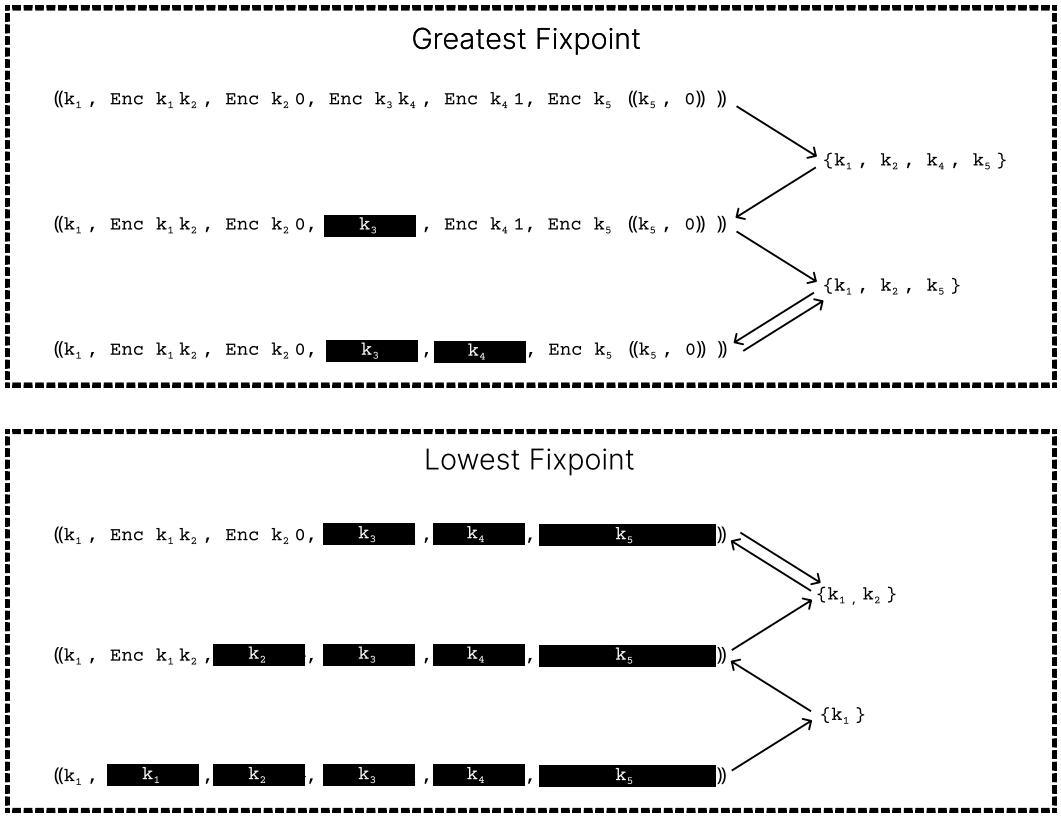

Fig. 6. Illustration of the greatest and least fixpoints of `keyRecovery`. Arrows denote the application of `hideEncrypted` and `extractKeys`. k_i denotes `VarK i` and $\boxed{k_j}$ denotes `Hidden(VarK i)`.

```
extractKeys (Enc (VarK 0) (VarK 1)) = {VarK 1}
```

The `VarK 0` is not extracted because it is only used to encrypt other data. On the other hand, `VarK 1` is extracted because, although encrypted, it actually appears in the expression. It is not hard to see (and to prove in Lean) that `keyRecovery` is a monotone function, so we can define `extractKeys` as its greatest fixpoint. Then, to compute the `adversaryView` of an expression, we use `hideEncrypted` with `adversaryKeys`. Finally, let us observe that using the greatest fixpoint allows the adversary to access the cyclically encrypted keys (see Figure 6), which, as briefly discussed in the examples, is the desired behaviour.

Variable Renaming. A *variable renaming* consists of two parts: a key variable renaming and a bit variable renaming. Key renaming is relatively straightforward: valid key renaming is a bijection that maps each key variable to another key variable. It is modeled as a function of type $\mathbb{N} \to \mathbb{N}$, and it is applied to an expression by applying it pointwise to every key variable. See Figure 7. `BitRenaming` is slightly more complex — it is a function that maps each bit variable to either another bit variable or its negation. (The ability to negate is essential because we want `VarB 0` and `Neg (VarB 0)` to be indistinguishable.) A bit renaming is *valid* if, when cast to a function of type $\mathbb{N} \to \mathbb{N}$, it becomes a bijection. Similar to key renaming, a bit renaming is applied to an

```
def KeyRenaming : Type := ℕ → ℕ

def validKeyRenaming (r : KeyRenaming) : Prop := Function.Bijective r

def applyKeyRenaming (r : KeyRenaming) (e : Expression s) : Expression s :=
  match e with
  | VarK n => VarK (r n) -- Rename `n` to `r n`.
  | Pair e1 e2 => Pair (applyKeyRenaming r e1) (applyKeyRenaming r e2)
  | Perm b e1 e2 => Perm b (applyKeyRenaming r e1) (applyKeyRenaming r e2)
  | Enc e1 e2 => Enc (applyKeyRenaming r e1) (applyKeyRenaming r e2)
  | e => e
```

Fig. 7. Definition of key renaming. (`Expression/SymbolicIndistinguishability.lean`.)

expression pointwise to every bit variable. (We omit the formal definition here; it can be found in file `Expression/SymbolicSemantics.lean`.)

Finally, a `varRenaming` is a pair consisting of a key renaming and a bit renaming. It is valid if both of its components are valid, and it is applied to an expression by independently applying the key and bit renamings.

Normalize. The function `normalizeExpr` simplifies expressions by applying the following rules (see Figure 8):

(1) `Neg (Bit 0)` simplifies to `Bit 1`, and `Neg (Bit 1)` simplifies to `Bit 0`.
(2) `Neg (Neg X)` simplifies to `X`.
(3) `Perm (VarB 0) X Y` simplifies to `(Y, X)`, and `Perm (VarB 1) X Y` simplifies to `(X, Y)`.
(4) `Perm (Neg B) X Y` simplifies to `Perm B Y X`.

Symbolic Indistinguishability. Finally, we combine the three parts to define the symbolic indistinguishability relation, by saying that two expressions e_1 and e_2 are symbolically indistinguishable, if we can rename the variables in e_1, in such a way that the two expressions are equal after normalizing and applying the adversary's view (see Figure 9).

3 EXTENDED EXAMPLE – CIRCUIT GARBLING

In this section, we present a use case for our framework: an implementation of circuit garbling in Lean, together with a formal proof of symbolic security. This example serves two primary purposes. First, it demonstrates the usefulness of the symbolic framework for formalizing cryptographic properties in Lean. Second, it provides a concrete symbolic proof that can serve as a starting point for further projects.

The structure of this section is as follows. In Section 3.1, we provide background on circuit garbling and multi-party computation, explaining their importance for cryptographic protocols. In Section 3.2, we present our modeling of the circuit garbling protocol and formally state its security property in the symbolic framework. Finally, in Section 3.3, we introduce a proof technique for reasoning about symbolic cryptography, which we developed during the formalization effort.

This work formalizes the results presented in [27], but modifies the proof to make it more suitable for proof assistants. This formalization effort led to two notable side results. First, we discovered that the use of pseudo-random generators (as employed in [27]) was unnecessary. We were able to simplify the construction to rely solely on encryption. This not only improves efficiency but

```
def normalizeB (p : BitExpr) : BitExpr :=
  match p with
  | Not (Not e) => normalizeB e
  | Not (Bit b) => Bit (not b)
  | e => e

def normalizeExpr {s : Shape} (p : Expression s) : Expression s :=
  match p with
  | BitE p => BitE (normalizeB p)
  | Perm (BitE b) p1 p2 =>
    let b' :=  normalizeB b
    let p1' := normalizeExpr p1
    let p2' := normalizeExpr p2
    match b' with
    | Bit b'' =>
      if b''
      then Pair p2' p1'
      else Pair p1' p2'
    | Not b'' =>
      Perm (BitE b'') p2' p1'
    | VarB k => Perm (BitE (VarB k)) p1' p2'
  | Pair p1 p2 => Pair  (normalizeExpr p1) (normalizeExpr p2)
  | Enc k e => Enc k (normalizeExpr e)
  | p => p
```

Fig. 8. Definition of normalization. (`Expression/SymbolicIndistinguishability.lean`.)

```
def symIndistinguishable (e1 e2 : Expression s) : Prop :=
  ∃ (r : varRenaming), validVarRenaming r ∧
   normalizeExpr (applyVarRenaming r (adversaryView e1)) = normalizeExpr
     (adversaryView e2)
```

Fig. 9. Definition of symbolic indistinguishability. (`Expression/SymbolicIndistinguishability.lean`.)

also simplifies the formalization effort. The second side result is the pseudo-fixpoint reasoning technique, which provides a structured approach to symbolic proofs of cryptographic security.

3.1 Multi-Party Computation and Circuit Garbling

Multi-party computations (MPCs) [20, 42] (see, also [18]) is a family of cryptographic protocols that allow multiple parties to jointly evaluate a function without revealing their inputs to each other. In this paper, we are interested in the *two*-party version of this notion. Specifically, consider a scenario in which two parties, Alice and Bob, wish to evaluate a publicly-know logical circuit where some input wires belong to Alice and others to Bob. The MPC protocol enables them to perform this evaluation without revealing anything more about their inputs than what can be inferred from the circuit's output. Two important properties of an MPC protocol are security and correctness. An MPC protocol is secure if it guarantees the privacy of the inputs, and correct if it ensures that the parties always reach the correct output. See [18] for a more detailed discussion of MPC, including the formal definitions of security and correctness.

One of the techniques for achieving secure multi-party computation is through Yao's circuit garbling [42]. The core of a garbling protocol (see also [27, Definition 5]) is a probabilistic function:

$$\text{Garble}(\mathit{circuit}, \mathit{input}) = (\mathit{gCircuit}, \mathit{gInput})$$

Here, *circuit* is a representation of a logical circuit (for a formal definition, see, e.g, [27, Definition 4]), and *input* is a bit vector representing the inputs to the circuit (the Garble procedure does not distinguish between input wires belonging to Alice or Bob). The output is a pair *gCircuit* and *gInput*, both encoded as bit vectors. The idea is that *gCircuit* and *gInput* should obscure the original circuit and input, in a way that allows for circuit evaluation ("correctness") while revealing nothing about the original input beyond what can be inferred from the output ("security").

Formally, the security of the garbling protocol is defined in terms of indistinguishability. In order for a garbling protocol to be secure (see also [27, Definition 6]), there must exist a function *Simulate*:

$$\text{Simulate}(\mathit{circuit},\ \mathit{output}) = (\mathit{sCircuit},\ \mathit{sInput})$$

Such that for every circuit C and input x, the following distributions are indistinguishable:

$$\text{Garble}(C, x) \quad \text{and} \quad \text{Simulate}(C, C(x))$$

Indeed, if it is possible to simulate the output of $\text{Garble}(C, x)$ without knowing the input x, then it is impossible to learn anything more about x from $\text{Garble}(C, x)$ than what can already be inferred from $C(x)$.

For the correctness of the garbling protocol (see also [27, Definition 6]), we require that there exists an explicit and efficient function *Evaluate*:

$$\text{Evaluate}(\text{gCircuit},\ \text{gInput}) = \text{output}$$

such that for every circuit C and input x, we have:

$$\text{Evaluate}(\text{Garble}(C, x)) = C(x)$$

In order for a garbling protocol to be usable in a two-party computation setting, it needs to be a *projective scheme*. To explain what this means, let us first define the function

$$\text{proj} : \left(T^2\right)^n \times \{0,1\}^n \to T^n,$$

which takes a list of pairs and a selector string (both of length n) and selects one element from each pair according to the corresponding bit in the selector string. We say that a garbling function is a *projective scheme* if it can be expressed as:

$$\text{Garble}(\text{circuit}, \text{input}) = (\text{gCircuit},\ \text{proj}(\text{gInputPairs}, \text{input})),$$

where *gCircuit* and *gInputPairs* are produced by a function that does not depend on the input:

$$\text{preGarble}(\text{circuit}) = (\text{gCircuit},\ \text{gInputPairs})$$

A secure projective garbling protocol can be used to implement a two-party computation protocol — see [11, Figure 3] or [11, Section 7.1]. For the sake of completeness, and to justify the notion of projective schemes, let us briefly sketch one way to achieve this. The construction requires a cryptographic primitive known as *oblivious transfer* [19, 38] (see [18]). Oblivious transfer works as follows: Alice has two values v_0 and v_1, and Bob has a bit $b \in \{0, 1\}$. With the oblivious transfer protocol, Bob learns v_b without learning the other value $v_{\neg b}$, while Alice does not learn b. See [34]. Since its introduction in the 1980s, oblivious transfer has been used as a building block for many cryptographic protocols. Several efficient implementations of oblivious transfer exist, based on multiple plausible assumptions, such as the *quadratic residuosity* [19, 38], *decisional Diffie-Hellman* [34], or *learning with errors* [36]. Using the oblivious transfer primitive, we can implement the two-party version of the garbling protocol as follows:

(1) Alice runs the *preGarble* function on C, obtaining *gCircuit* and *gInputPairs*.
(2) Alice sends *gCircuit* to Bob.
(3) For each bit in her input, Alice sends the corresponding value from *gInputPairs* to Bob.
(4) For each bit in his input, Bob uses oblivious transfer to receive from Alice the corresponding value from *gInputPairs*.
(5) Bob now has *gCircuit* and *gInput*, so he can evaluate the circuit and send the output to Alice.

Let us finish this section by noting that garbling circuits is an important and active area of research. We refer the reader to [11] for a survey of protocols and applications. One particularly active line of work focuses on optimizing the garbling protocol to make it more efficient, e.g., the *free XOR* [25] and the *row-reduction* [37] techniques. Interestingly, the original paper on circuit garbling by Yao [42] did not provide a formal proof of security, and it took over two decades before a (pen-and-paper) proof of security was provided by Lindell and Pinkas [29] and by Bellare et al. [11]. This may serve as an illustration of the complexity of the problem, and the need for formalizing security proofs in a proof assistant.

3.2 Security Proof of Circuit Garbling

In this section, we explain how modelled the circuit garbling protocol in Lean, and how we formalized its symbolic security statement. However, since our implementation is based on the implementation provided by [27] (with the pseudo-random generator removed), we will not repeat it here. Similarly, since [27] already provides a pen-and-paper proof of symbolic security, we will not present our formalization of the proof here either. Instead, we focus on presenting its statement in Lean. For further reading, we refer the reader to [27] or to our implementation.

We implemented `preGarble` in Lean and used it to define `Garble`, as required by the definition of a *projective scheme*. Importantly, `Garble` does not produce a distribution over bit vectors directly. Instead, it returns a cryptographic expression, which can later be compiled into a distribution using the computational semantics introduced in the previous section. (Since cryptographic expressions do not support lists, we represent lists using nested tuples instead.) We also implemented the function `SimulateG`, which simulates the output of `Garble` given only the circuit and its output; this function also returns a cryptographic expression. Finally, we proved that for every circuit C and input x, the expressions `Garble C x` and `SimulateG C (evalCircuit C x)` are symbolically indistinguishable. See Figure 10 for the definition of `Garble` (in terms of `preGarble`), the types of `preGarble` and `SimulateG`, and the statement of the symbolic indistinguishability property.

3.3 Proof Technique for Working with Fixpoints

Let us start by briefly outlining our proof technique for establishing symbolic indistinguishability of garbled circuits. Let g be the expression produced by `Garble` and s the one produced by `SimulateG`. We begin the proof by giving explicit definitions for `adversaryKeys g` and `adversaryKeys s`. (This is the most difficult step, as it involves reasoning about fixpoints.) The remaining steps are relatively straightforward: we use the definitions of `adversaryKeys` to compute explicit formulas for `adversaryView` of g and s, and then provide a variable renaming r such that applying r to `adversaryView g`, followed by normalization, yields `adversaryView s`. The main technical challenge is reasoning about the `adversaryKeys` function, which is defined via a fixpoint computation.

In the remainder of this section, we explain our general technique for reasoning about fixpoints. To illustrate it, consider an expression `e = (`$\mathtt{e}_1$`, `$\mathtt{e}_2$`)`. The core difficulty lies in the fact that `adversaryKeys` is not compositional: computing it separately for $\mathtt{e}_1$, $\mathtt{e}_2$, and $\mathtt{e}_3$ tells us little about `adversaryKeys e`. For example, suppose $\mathtt{e}_1$ is the key variable k_1, and $\mathtt{e}_2$ `= Enc` $\mathtt{k}_1$ $\mathtt{k}_2$.

```
def preGarble (c : Circuit inputBundle outputBundle) :=
  (garbledCircuitType c) × (projectionLabelType inputBundle) := [...]

def Garble (c : Circuit inputBundle outputBundle) (input : bundleBool inputBundle)
  : Expression (Shape.PairS (garbledCircuitShape c) (garbledInputShape inputBundle)) :=
  let (garbledCircuit, inputPairs) := preGarble c
  garbleOutputToExpr (garbledCircuit, makeProjection inputPairs input)
  -- `garbleOutputToExpr` recursively transforms Lean-pairs (x, y) to Expression-pairs
    ((x, y))

def SimulateG (c : Circuit inputBundle outputBundle) (output : bundleBool outputBundle)
  : Expression (Shape.PairS (garbledCircuitShape c) (garbledInputShape inputBundle)) :=
    [...]

lemma garblingSecureSymbolic :
    ∀ (c : Circuit inputBundle outputBundle) (input : bundleBool inputBundle),
        symIndistinguishable (Garble c input) (SimulateG c (evalCircuit c input))
```

Fig. 10. Types of the garble function. (`Garbling/GarbleDef.lean`)

Then `adversaryKeys` e_1 = $\{k_1\}$ and `adversaryKeys` $e_2 = \emptyset$, but `adversaryKeys` $e = \{k_1, k_2\}$, with k_2 appearing seemingly from nowhere.

However, in practice, expressions like those produced by `Garble`, `SimulateG`, and most reasonable protocols are highly structured. This allows us to use a powerful fixpoint reasoning technique. First, those experrsions do not contain cyclic encryptions, so we can prove that `keyRecovery` has a unique fixpoint, and not worry about `adversaryKeys` being the greatest one.

Another key property of g and s is that they are usually non-overlapping: if $(\!(e_1, e_2)\!)$ is a subexpression of g or s, then the sets of keys written in e_1 and e_2 (i.e. the ones returned by `extractKeys`) are disjoint — although e_2 may still use keys from e_1 for encryption. To take advantage of this, we introduce the following notion: a set of keys K is a *pseudo-fixpoint with respect to* U *on* e if

$$\texttt{(keyRecovery e K)} \cap \texttt{U} = \texttt{K} \cap \texttt{U}$$

That is, the keys in K agree with the result of `keyRecovery` on the subset U.

We use this definition as follows. To prove that a set X is a fixpoint of `keyRecovery` on $(\!(e_1, e_2)\!)$, it suffices to show that X is a pseudo-fixpoint on e_1 with respect to `extractKeys` e_1, and on e_2 with respect to `extractKeys` e_2. Of course, this only works when $(\!(e_1, e_2)\!)$ satisfies the sortedness condition above. Interestingly, this technique turned out to be powerful enough to reason about `adversaryKeys` for g and s, significantly simplifying the symbolic indistinguishability proof.

4 COMPUTATIONAL INDISTINGUISHABILITY

In this section, we revisit the notion of indistinguishability from the standard computational perspective. We begin by discussing the texbook IND-CPA (indistinguishability under chosen-plaintext attack) definition of encryption security and how we formalize it in Lean. Next, we formally state and prove the computational soundness theorem, which establishes a connection between symbolic and computational indistinguishability. This theorem enables us to lift symbolic indistinguishability results to the computational setting; in particular, it allows us to derive the computational security of garbled circuits from their symbolic proof.

```
def encryptionScheme : Type := (κ : ℕ) -> encryptionFunctions κ

-- We lift `exprToDistr` to work with `encryptionScheme` and produce a family of
    distributions
def exprToFamDistr (enc : encryptionScheme) (e : Expression s) :=
    fun κ => exprToDistr (enc κ) e
```

Fig. 11. Definition of an encryption scheme. (`Expression/ComputationalSemantics/encryption-IndCpa.lean`)

4.1 Indistinguishability-Based Security of Encryption Schemes

The standard notion of encryption security is called *IND-CPA*, which stands for *indistinguishability under chosen-plaintext attack*. This definition captures the idea that an adversary should not learn anything about the plaintext, even when the same encryption key is used to encrypt multiple messages chosen by the adversary. IND-CPA security can be formalized in various equivalent ways; here, we follow the definition given in [27, Definition 2], which defines it in terms of two oracles, O_L and O_R. Both oracles are initialized with a randomly chosen key k, and respond to queries of the following form:

Input:	A pair of messages of the same length (m_L, m_R)
Output of O_L:	m_L encrypted with k
Output of O_R:	m_R encrypted with k

We say that an encryption scheme is *IND-CPA secure* if no probabilistic polynomial-time (PPT) adversary can distinguish between the two oracles with non-negligible probability. However, the definitions of polynomial time and negligible probability both require a parameter that measures the complexity of the oracles. This parameter is known as the *security parameter*[6] κ, which in our case is equal to the length of the encryption key. For this reason, we define an encryption scheme as a *family* of encryption functions indexed by the security parameter κ (see Figure 11). Based on such an encryption scheme, we construct the families of oracles O_L^κ and O_R^κ, and say that the encryption scheme is IND-CPA secure if no PPT adversary can distinguish the outputs of the two oracles with non-negligible probability, both with respect to κ. (A probability is called *negligible* if it decreases faster than the inverse of any polynomial in κ.)

4.1.1 Oracles in Lean – VCVio. To talk about *oracles* and *oracle computations* in Lean, we use the VCVio library[7] [40]. In our formalization, O_L and O_R are implemented as VCVio oracles, and the adversary is modeled as an oracle computation that interacts with them. Let us now briefly discuss how VCVio defines the two notions. The interface between oracles and oracle computations is defined by the oracle's specification, `OracleSpec I`. It is parameterized by a type `I`, which represents the set of query labels that can be asked to the oracle. For each $i \in I$, the specification assigns a pair of types:

$$(\text{Input type of the query } i,\ \text{Output type of the query } i)$$

As an example, let us discuss the specification of the oracles O_L and O_R. (Observe that since the two oracles answer the same type of query, they must share the same specification.) Although they intuitively implement only one type of query, for technical reasons, it is convenient to take $I = \mathbb{N}$ and define a separate query for each possible message length. The n-th query then inputs a pair of

[6] In the literature, the security parameter is often denoted by λ. However, since in Lean λ is a reserved keyword, we denote it by κ instead.

[7] See: github.com/dtumad/VCV-io.

messages of length n and returns a ciphertext of length `encLength n`. The specification is formally defined as `oracleSpecIndCpa` in Figure 14.

An *oracle* is defined as an implementation of the oracle specification and has the type `QueryImpl (spec : OracleSpec I) M`, which, to every $i \in I$, associates a function of type:

$$\text{In } i \to M\ (\text{Out } i)$$

where In i and Out i are the input and output types of the query i, as specified by `spec`. (For example, in the case of the oracles O_L and O_R, In n is `BitVector n × BitVector n`, and Out n is `BitVector (encLength n)`.) Observe that the output type of the oracle query is modified by a functor M — this should be a monad that represents the computational effects used by the oracle, such as state or nondeterminism. In our case, we always use `M x = PMF (Option x)`, which models probabilistic computations that may fail. (This monad is also commonly written as `(OptionT PMF) x`.) We use the `PMF` monad to model the probabilistic nature of encryption schemes, while the `Option` monad is required by VCVio, which assumes that oracle computations can fail. (However, the use of `Option` is purely technical: we will only consider oracles that never fail, i.e., return `none` with probability zero.)

Finally, an oracle computation that interacts with an oracle and returns a value of type `X` is represented by the type `OracleComp (spec : OracleSpec I) X`. It can be seen as an abstract computation that is designed to work with any oracle implementing the specification `spec` (formally, it is defined as a *free monad*, see e.g. [39, Section 6]). It comes equipped with a function `simulateQ` that, given an actual oracle implementation `impl : QueryImpl spec M`, performs the computation and returns a value of type `M X`.

4.1.2 Oracles' indistinguishability in Lean. We are now going to define computational indistinguishability for *families of seeded oracles* — a notion we will use to formalize IND-CPA security of encryption schemes. A *family of seeded oracles* (formally defined in Figure 12) consists of one oracle for each security parameter κ, and each oracle is initialized with a random value drawn from a specified distribution. For example, the oracles O_L and O_R are initialized with a uniformly random encryption key.

An adversary that interacts with a family of seeded oracles is represented as a *family of oracle computations* (formalized as `famOracleComp` in Figure 12). Such a family consists of one oracle computation for each value of the security parameter κ. Since we want the adversary to be a probabilistic computation, it is, in addition to its input oracle, also given access to a randomness oracle, which generates random values from specified distributions.

An important property of families of seeded oracles and families of oracle computations is that they can be plugged together using `simulateQ`, producing a family of distributions over output values (parametrized by κ). This plugging function is formalized as `compToDistrGen` in Figure 12.

We say that two families of seeded oracles are *computationally indistinguishable* if no probabilistic polynomial-time distinguisher can tell them apart with non-negligible probability. A distinguisher is a special kind of family of oracle computations that returns a Boolean for all values of the security parameter κ (i.e., `famOracleComp Spec (fun _ => Bool)`). Formalizing the definition of negligibility is straightforward (see `negl` in Figure 13). The most challenging part is defining which families of oracle computations run in polynomial time — we postpone this discussion to Section 4.2.2. For now, the definition of indistinguishability simply takes a predicate `(IsPolyTime : PolyFamOracleCompPred)` as a parameter. See Figure 13 for the formal definition.

With the notion of indistinguishability of families of seeded oracles in place, we can now formally define the IND-CPA security of an encryption scheme by directly formalizing the definition given in Section 4.1. See Figure 14.

```
-- A family of seeded oracles is parametrized by a family of specifications.
structure famSeededOracle (Spec : ℕ -> OracleSpec I) where
  Seed : (κ : ℕ) -> Type -- The type of the seed
  seedDistr : (κ : ℕ) -> OptionT PMF (Domain κ) Seed -- The distribution of the seed
  queryImpl : (κ : ℕ) -> Seed κ -> QueryImpl (Spec κ) (OptionT PMF) -- Implementation
    of the queries

-- A family of oracle computations is also parametrized by a family of specifications.
-- The function `withRandomI` adds randomness queries to the specification
def famOracleComp (Spec : ℕ -> OracleSpec I) (Output : ℕ -> Type) :=
    (κ : ℕ) -> OracleComp (withRandomI Spec κ) (Output κ)

-- A function that runs a given `famOracleComp` on a given `famSeededOracle`.
-- The function `addRandom` provides the implementations of the randomness queries
-- introduced by `withRandomI`.
def compToDistrGen (oracle : famSeededOracle Spec) (comp : famOracleComp Spec Output)
    (κ : ℕ)
  : (OptionT PMF (Output κ)) := do
    let seed ← oracle.seedDistr κ
    OracleComp.simulateQ (addRandom (oracle.queryImpl κ seed)) (comp κ)
```

Fig. 12. Definition of a family of seeded oracles. (`ComputationalIndistinguishability/Def.lean`)

4.2 Computational Soundness Theorem

Having formalized the IND-CPA security assumption for the encryption scheme, we are now ready to formalize the computational soundness theorem, which states that symbolic indistinguishability implies computational indistinguishability (as long as the encryption scheme used in the computational semantics is IND-CPA secure). In this subsection, we will: (a) formalize symbolic indistinguishability for families of distributions; (b) discuss our axiomatic approach for handling polynomial-time computations; (c) state the computational soundness theorem; and (d) explain our approach to proving it.

4.2.1 Computational Indistinguishability of Distributions. The computational soundness theorem is stated in terms of the indistinguishability of families of distributions, rather than of families of seeded oracles. Because of this, we begin by formalizing this notion of indistinguishability. Although it is quite similar — and simpler — than the definition of oracle indistinguishability, both notions will play a role in our axiomatic approach to polynomial-time computations.

A *family of distributions* is a function `(κ : ℕ) → PMF (Option (Domain κ))`, where `Domain` is a family of types. The adversary is modeled as a family of simple computations, consisting of functions `Domain κ → PMF (Output κ)` for each κ. To apply a family of computations f to a family of distributions d, we sample a value x from d_κ for each κ, and compute $f_\kappa(x)$. See Figure 15.

Instead of introducing a new abstract predicate for polynomial-time families of simple computations, we reuse the one defined for families of oracle computations. To do this, we observe that every family of simple computations f_κ can be lifted to a family of oracle computations that queries a single oracle for a value of type `Domain κ`, applies f_κ to this value, and then uses the randomness oracle to sample from the resulting distribution. Thanks to this lift, we can apply any abstract predicate `p : PolyFamCompPred` defined for families of oracle computations to families of simple computations. See Figure 16 for details.

```
-- A type of predicates on `famOracleComp`.
-- Used to represent the polynomial-time predicate.
def PolyFamOracleCompPred : Type :=
    famOracleComp Spec Output -> Prop

-- Definition of negligible probability.
-- `NNReal` represents non-negative real numbers.
def negl (f : ℕ -> NNReal) : Prop :=
  ∀ k, ∃ (B : ℝ), ∀ i, (f i) * (i^k) <= B

-- Advantage is the difference between probabilities
-- that a distribution returns `True`. It is used to measure
-- how well did the adversary distinguish two oracles
def advantage (x y : ℕ → PMF (Option Bool)) : ℕ → NNReal :=
  fun κ =>
    -- `distance p q` is defined as | p - q |.
    distance (getPMF (x κ) (some True)) (getPMF (y κ) (some True))

def CompIndistinguishabilitySeededOracle
  (IsPolyTime : PolyFamOracleCompPred)
  (o1 o2 : famSeededOracle Spec)
  : Prop :=
    -- All distinguishers ...
    ∀ distinguisher : famOracleComp Spec (fun _ => Bool),
    -- ... that run in polynomial time ...
    (IsPolyTime distinguisher) ->
    -- ... only achieve negligible advantage.
    negl (advantage (compToDistrGen o1 distinguisher) (compToDistrGen o2
     distinguisher))
```

Fig. 13. Definition of oracle indistinguishability. (`ComputationalIndistinguishability/Def.lean`)

Finally, we say that two families of distributions are indistinguishable if no polynomial-time distinguisher (modeled as a family of simple computations $d_\kappa : \mathtt{Domain}_\kappa \to \mathtt{PMF\ Bool}$) can distinguish them with non-negligible probability. See Figure 17.

4.2.2 Axiomatizing Polynomial Time. The notion of polynomial-time computation is central to much of modern cryptography, but it is challenging to model in formal theorem provers such as Lean (or Easycrypt). This is because they do not have a built-in notion of a function's running time, so polynomial time must be defined from scratch using Turing machines or some other equivalent model. Dealing with *probabilistic polynomial-time adversaries* is particularly challenging, as they are usually framed as *probabilistic* and *interactive* Turing machines — a fairly natural but impractical model for formal verification. For this reason, in this work we adopt an alternative, axiomatic approach and model it as an abstract predicate on families of oracle computations (i.e., of type `PolyFamOracleCompPred`) that satisfies the following two axioms:

1. It is closed under composition. Observe that families of oracle computations (`famOracleComp`) can be composed with families of simple computations (`famComp`), resulting in a new element of `famOracleComp`. This is because, just before returning its result, a `famOracleComp` can apply a `famComp` transformation to it. This axiom states that if both the input `famOracleComp` and the input `famComp`

```
def oracleSpecIndCpa (κ : ℕ) (enc : encryptionFunctions κ) : OracleSpec ℕ :=
  fun n => ((BitVector n) × (BitVector n), BitVector (enc.encryptLength n))

-- A type used to index the oracles O_l and O_r.
inductive Side : Type
  | L
  | R

def choose (s : Side) (x : X × X) : X :=
  match s with
  | L => x.1
  | R => x.2

-- Implementation of an oracle query: encrypt the left or right message based on w
def indCpaOracleImpl (s : Side) (κ : ℕ) (enc : encryptionFunctions κ)  (key :
    BitVector κ) :
    QueryImpl (oracleSpecIndCpa κ enc) (OptionT PMF) := {
        impl query :=
            -- Unpack the query ...
            let OracleSpec.query l ⟨msg₁, msg₂⟩ := query
            -- ... and encrypt the adequate message.
            enc.encrypt key (choose s (msg₁, msg₂))
    }

-- Pack `indCpaOracleImpl` into a `famSeededOracle`.
def seededIndCpaOracleImpl (s : Side) (enc : encryptionScheme) :
    famSeededOracle (fun κ ↦ oracleSpecIndCpa κ (enc κ)) := {
        Seed κ := BitVector κ,
        --^ Key with κ bits.
        seedDistr κ := PMF.uniformOfFintype (BitVector κ),
        --^ Draw the key uniformly at random.
        queryImpl κ key := indCpaOracleImpl s κ (enc κ) key
        --^ Encrypt the left or the right msg based on s.
    }

def encryptionSchemeIndCpa (IsPolyTime : PolyFamOracleCompPred) (enc :
    encryptionScheme)  : Prop :=
  CompIndistinguishabilitySeededOracle IsPolyTime (seededIndCpaOracleImpl Side.L enc)
    (seededIndCpaOracleImpl Side.R enc)
```

Fig. 14. Definition of IND-CPA security of an encryption scheme. (`Expression/Computational-Semantics/encryptionIndCpa.lean`)

run in polynomial time, then the resulting `famOracleComp` also runs in polynomial time. See Figure 18 for the formal statement.

2. It contains specific computations. The second axiom is more ad hoc than closure under composition: it postulates that a specific `famOracleComp`, called `removeOneKeyReduction`, which we use in the proof, runs in polynomial time. It is defined in Figure 20, and we explain why it runs in polynomial

```
-- We consider two kinds of families of distributions:

-- The possibly failing ones, which are more compatible with VCVIO.
def famDistr (Domain : ℕ -> Type) := (κ : ℕ) -> OptionT PMF (Domain κ)

-- A family of computations used to transform both `famDistr` and `famDistr`.
def famComp (Input : ℕ -> Type) (Output : ℕ -> Type) :=
    (κ : ℕ) -> (Input κ) -> (PMF (Output κ))

-- A family of computation `comp` is applied to a family of distributions `input`
-- by drawing one value from `input` and applying `comp` to it.
def compToDistrSimple (input : famDistr Input) (comp : famComp Input Output) (κ : ℕ)
    : OptionT PMF (Output κ) := do
        -- Draw a single random value from the input distribution ...
        let z ← input κ
        -- ... and apply the `famComp` to the result.
        o κ z
```

Fig. 15. Definition distribution indistinguishability. (`ComputationalIndistinguishability/Def.lea`)

```
-- Lift `famComp` to `famOracleComp`.
def simpleCompAsGenComp (f : famComp Input Output)
  : famOracleComp (fun κ => simpleCompSpec (Input κ)) Output :=
  fun κ => do
    -- Ask the oracle for the input argument `x` ...
    let (x : Input κ) ← getInputArg Input κ
    -- ... apply `f` to `x` ..
    let distr := f κ x
    -- .. and sample from the resulting distribution.
    let ret ← sample (f κ x)
    return ret

-- Recast `(polyTime : PolyFamOracleCompPred)` to work with define `polyTimeFamComp`.
def polyTimeFamComp (polyTime : PolyFamOracleCompPred) (o : famComp Input Output) :
    Prop :=
  polyTime (simpleCompAsGenComp o)
```

Fig. 16. Lifting the polynomial-time predicate from families of oracle computations to families of distributions. (`ComputationalIndistinguishability/Def.lea`)

time on page 24. This axiom serves as a way of lifting a pen-and-paper complexity analysis into a formal framework, while relying on the reader to verify this particular complexity claim.

4.2.3 Computational Soundness Theorem. We are now ready to state the computational soundness theorem. It asserts that symbolically indistinguishable expressions yield computationally indistinguishable distributions — provided the underlying encryption scheme is IND-CPA secure. See Figure 19 for the formal statement in Lean. We start by unfolding the definition of symbolic indistinguishability, obtaining that for some valid renaming `r`, we have:

```
normalizeExpr (applyVarRenaming r (adversaryView expr1)) =
```

```
-- Now, we can lift the notion of polynomial time of oracle computations
-- to `famComp`s. This is because every function of type `X -> PMF Y`
-- can be seen as an `OracleComp` that gets its input trough an oracle.
-- This lift is implemented as `simpleCompAsGenComp` omitted in this listing.
def polyTimeFamComp (polyTime : PolyFamOracleCompPred) (o : famComp Input Output) :
    Prop :=
  polyTime (simpleCompAsGenComp o)

-- We are now ready to define indistinguishability
def CompIndistinguishabilityDistr
  (IsPolyTime : PolyFamOracleCompPred)
  (distr1 distr2 : famDistr Domain) : Prop :=
  -- All distinguishers...
  forall distingisher : famComp Domain (fun _ => Bool),
  -- ... that run in polynomial time ...
  (polyTimeFamComp IsPolyTime distingisher) ->
  -- ... only achieve negligible advantages.
  negl (advantage (compToDistrSimple distr1 distingisher) (compToDistrSimple distr2
    distingisher))
```

Fig. 17. Definition of indistinguishability of families of distributions. (Computational-Indistinguishability/Def.lea)

```
def composeOracleCompWithSimpleComp
  (oracleComp : famOracleComp Spec Domain)
  (f : famComp (Domain) Output)
  : famOracleComp Spec Output := fun κ => do
    let val : (Domain κ) ← oracleComp κ
    let ret ← sample (f κ val)
    return ret

-- The composition axiom about `isPolyTime`
def PolyTimeClosedUnderComposition (isPolyTime : PolyFamOracleCompPred) : Prop :=
  forall I (Spec : (κ : ℕ) -> OracleSpec I) Domain Output,
  forall (oracleComp : famOracleComp Spec Domain) (f : famComp Domain Output),
  isPolyTime oracleComp ->
  polyTimeFamComp isPolyTime f ->
  isPolyTime (composeOracleCompWithSimpleComp oracleComp f)
```

Fig. 18. The composition axiom for polynomial time. (ComputationalIndistinguishability/Def.lean)

`normalizeExpr (adversaryView expr2)`

We then show that both `normalizeExpr` and `applyVarRenaming r` (for a valid `r`) preserve the computational semantics — that is, they do not affect the resulting distributions:

- The proof for `normalizeExpr` proceeds by a rather straightforward structural induction on expressions.
- The proof for `applyVarRenaming r` relies on two observations: (a) variable renaming can be simulated by permuting the corresponding values before evaluation, and (b) shuffling a uniformly random vector of values does not alter the resulting distribution.

```
theorem symbolicToSemanticIndistinguishability :
  -- For every predicate `polyTime`,
  (polyTime : PolyFamOracleCompPred)
  -- that satisfies our axioms for polynomial computations
  (HPolyTime : PolyTimeClosedUnderComposition IsPolyTime)
  (Hreduction : forall enc shape (expr : Expression shape) key₀, IsPolyTime
    (removeOneKeyReduction enc expr key₀))
  -- For every encryption scheme,
  (enc : encryptionScheme)
  -- that is IND-CPA secure (with respect to `polyTime`).
  (HEncIndCpa : encryptionSchemeIndCpa IsPolyTime enc)
  -- For every pair of expressions,
  (expr1 expr2 : Expression shape)
  -- that is symbolically indistinguishable
  (Hi : symIndistinguishable expr1 expr2) :
  -- it is also computationally indistinguishable
  CompIndistinguishabilityStrict IsPolyTime
    (exprToFamDistr enc expr1) (exprToFamDistr enc expr2)
```

Fig. 19. The statement of the computational soundness theorem. (`Expression/Computational-Semantics/Soundness.lean`).

Since computational indistinguishability is both transitive and reflexive, it remains only to show that the distributions produced by `expr` and `adversaryView expr` are computationally indistinguishable. In the following few paragraphs, we explain how to prove this.

We start by introducing the operation `removeOneKey`, which takes an expression `e` and a single key `k`, and replaces all occurrences of `Enc k X` with `Hidden k`. Next, we observe that we can reach `adversaryView expr` by repeatedly applying `removeOneKey` to `expr`, using keys that are not written in the expression—i.e., not returned by `extractKeys`. (Note that as we proceed, `extractKeys` might return fewer keys, making more keys available for removal.) Thanks to this observation, we can apply transitivity once again, reducing the proof to showing that if `k ∉ extractKeys expr`, then the distributions produced by `expr` and `removeOneKey expr k` are computationally indistinguishable.

In order to prove this, we make use of the following observation: Let $\mathtt{O}_1$ and $\mathtt{O}_2$ be two computationally indistinguishable families of seeded oracles, and let `F` be a family of oracle computations, then the distributions $\mathtt{D}_1$ `= compToDistrGen` $\mathtt{O}_1$ `F` and $\mathtt{D}_2$ `= compToDistrGen` $\mathtt{O}_2$ `F` are also computationally indistinguishable. Indeed, if $\mathtt{D}_1$ and $\mathtt{D}_2$ were distinguishable, then the adversary could distinguish the oracles $\mathtt{O}_1$ and $\mathtt{O}_2$ by first running `F` and then the simple computation that distinguishes $\mathtt{D}_1$ and $\mathtt{D}_2$. (Here we use the axiom that polynomial time oracle computations are closed under compositions with polynomial time simple computations.) This observation is formalized as `lemma reduction` in `ComputationalIndistinguishability/Lemmas.lean`.

Equipped with this observation, we can now show that `expr` and `removeOneKey expr k` are indistinguishable, provided that `k ∉ extractKeys expr`. For this, we will define a family of oracle computations `removeOneKeyReduction` that, when run on the oracles O_L and O_R from the definition of IND-CPA security, will produce respectively `expr` and `removeOneKey expr k`. Most of the time, `removeOneKeyReduction` behaves like the computational semantics (i.e. `exprToDistr`). However, when it encounters an expression encrypted with the key that is being removed (i.e. `Enc k X`), it does not encrypt it directly, but instead sends the following pair of messages to the oracles:

(Bit vector corresponding to X, Bit vector of 1's)

```
-- This is the subroutine used to encrypt a message.
def encryptPMFOracle (enc : encryptionFunctions κ) (key₀ : ℕ)
    (kVars : (ℕ -> BitVector κ)) (key : ℕ) (input : BitVector d) :=
    -- Check if we encrypt with `key₀` (i.e. the key that is being removed) ...
    if key = key₀ then
      -- ... if so, then we query the oracle on the pair `(input, ones)`.
      innerQuery (oracleSpecIndCpa κ enc) d (input, ones)
    else
      -- ... if not, we encrypt input directly using `key`.
      sample (enc.encrypt (kVars key) input)

-- We define the analog of `evalExpr`, but using `encryptPMFOracle` to encrypt.
-- Except of `Enc`, it behaves the same as `evalEnc`.
def reductionToOracle (enc : encryptionFunctions κ) (kVars : (ℕ -> BitVector κ))
    (bVars : ℕ -> Bool) (e : Expression shape) (key₀ : ℕ) :=
    match e with
    | Enc (VarK k) e => do
        -- Instead of directly encrypting, we call `encryptPMFOracle`.
        let e' ← reductionToOracle enc kVars bVars e key₀
        encryptPMFOracle enc key₀ kVars k e'
    [...]
    -- (Other cases are the same as in `evalExpr` except that the recursive calls
    -- are made to `reductionToOracle` and not `evalExpr`).

-- Finally, we feed `reductionToOracle` with uniformly random variables.
def removeOneKeyReduction (enc : encryptionScheme) (e : Expression shape) (key₀ : ℕ) :=
  fun κ => do
      let l := (getMaxVar e + 1)
      let bVars ← sample (PMF.uniformOfFintype (Fin l -> Bool))
      let kVars ← sample (PMF.uniformOfFintype (Fin l -> BitVector κ))
      reductionToOracle (enc κ) (extendFin ones kVars) (extendFin false bVars) e key₀
```

Fig. 20. Definition of `removeOneKeyReduction`. (`Expression/ComputationalSemantics/SoundnessProof/HidingOneKey.lean`)

Now the left oracle $\mathtt{O}_l$ will return the encryption of `X` with a uniformly random key, which corresponds to the semantics of `Enc k X`. On the other hand, the right oracle $\mathtt{O}_r$ will return the encryption of a vector of ones, which corresponds to the semantics of `Hidden k`. Moreover, both oracles will consistently use the same key `k` (chosen uniformly at random) to answer all the queries. (It is important that `k ∉ extractKeys expr`, as we cannot extract `k` from the oracles, so we would not be able to output it.) It follows that `compToDistrGen` $\mathtt{O}_l$ `removeOneKeyReduction` will produce the same distribution as `expr`, and `compToDistrGen` $\mathtt{O}_r$ `removeOneKeyReduction` will produce the same distribution as `removeOneKey expr k`. Since the oracles are indistinguishable (by the IND-CPA assumption), we can conclude that `expr` and `removeOneKey expr k` result in computationally indistinguishable distributions (as long as `k ∉ extractKeys expr`), thus concluding the proof of the computational soundness theorem.

Complexity analysis of `removeOneKeyReduction`. Let us conclude this section by discussing the complexity of `removeOneKeyReduction`, whose formal definition is given in Figure 20, to justify the second axiom of the polynomial time predicate `PolyFamOracleCompPred`. (Actually, it is pretty

intuitive that `removeOneKeyReduction` runs in polynomial time, but since this is the *weakest link* of our proof, let us provide a formal pen-and-paper complexity analysis.)

Fix an encryption scheme `enc` that runs in polynomial time[8]; that is, it encrypts a message of length `n` using a key of length κ in time bounded by $p(n+\kappa)$ for some polynomial p. Let us now show that for every fixed expression `e` (and removed key $\texttt{key}_0$), the computation `removeOneKeyReduction` runs in time polynomial in κ. We begin by noticing that `removeOneKeyReduction` (as defined in Figure 20) consists of three parts:

(1) Draw at most $|e|$ random bits. This can be done in $|e|$ coin tosses (i.e., constant time with respect to κ).
(2) Draw at most $|e|$ random keys. This can be done in $|e| \cdot \kappa$ coin tosses (i.e., linear time with respect to κ).
(3) Run `reductionToOracle`, defined in Figure 20.

The first two steps are clearly polynomial in κ, so we only need to analyze the third step. Let us start by analyzing the length of the bit vector produced by `reductionToOracle`.

Lemma 4.1. *For a fixed expression* `e`*, the length of* `removeOneKeyReduction e` *is polynomial in* κ*.*

Proof. It is not hard to see that the length of the bit vector produced by `reductionToOracle` depends only on the shape of e (similarly to `evalExpr`). We prove the lemma by induction on the shape of e:

- For the base cases $s = \mathbb{B}$ and $s = \mathbb{K}$, the lengths are 1 and κ, respectively, both of which are polynomial in κ.
- For the case $s = ((s_1, s_2))$, the length of the bit vector is the sum of the lengths of the bit vectors produced by s_1 and s_2. By the induction hypothesis, both are polynomial in κ, so the total is also polynomial.
- Finally, consider the case $s = \texttt{EncS}\ s'$. Since $\texttt{encrypt}(k, n)$ runs in time $p(n + \kappa)$, the length of its output is also bounded by $p(n + \kappa)$. Here, n is the length of the bit vector produced by s', which by the induction hypothesis is bounded by $q(\kappa)$ for some polynomial q. It follows that the final output length is bounded by $p(q(\kappa) + \kappa)$, which is polynomial in κ.

□

Now, to analyze the time complexity of `reductionToOracle`, observe that each of its steps runs in time bounded by $p(n + \kappa)$, where n is the length of the intermediate result — this is because encryption is the most expensive operation (other steps run in linear time). Since the lengths of the intermediate results are bounded by the length of the final output, we know that each step runs in time bounded by $p(q(\kappa) + \kappa)$. Moreover, since there are at most $|e|$ steps, it follows that the total running time of `reductionToOracle` is bounded by $|e| \cdot p(q(\kappa) + \kappa)$, which is polynomial in κ. This concludes the proof that `removeOneKeyReduction` runs in polynomial time for every fixed expression `e`.

Let us finish this analysis by observing that, in principle, the exponent of the running time of `removeOneKeyReduction` depends on $|e|$ (which is not a problem, since e is fixed). However, if we assume — as is often the case in practice — that the encryption scheme produces outputs of length linear in $\kappa + n$, then the exponent becomes independent of $|e|$ and equal to the degree of p.

[8]The assumption that an encryption scheme runs in polynomial time does not actually follow from IND-CPA security, but since it is a common assumption in cryptography (exponential-time encryption schemes are not very useful), we will use it here.

5 CONCLUSION AND FUTURE WORK

We have presented a formalization of the symbolic cryptography framework using Lean, including a computational soundness theorem for symbolic indistinguishability. We have also demonstrated that this framework can be used to prove fairly complex properties of cryptographic protocols, such as the security of circuit garbling. We hope to have convinced the reader that the symbolic framework offers a clean, intuitive, and aesthetically pleasing way to reason about cryptographic properties. Below, we outline a few directions for further development of this framework:

1. Analysing more protocols. The most immediate next step would be to prove the security of more cryptographic protocols using the symbolic framework. One exciting direction is to formalize the security of optimization techniques for garbled circuits, such as *free XOR* [25] and *row reduction* [37]. However, most of these techniques would require an extended language of symbolic expressions.

2. Extending the language of cryptographic expressions. This leads to a second direction: extending the symbolic expression language to support additional cryptographic primitives, such as pseudo-random generators or key xor operations. A pen-and-paper version of a computationally sound symbolic framework with pseudo-random generators already appears in [27], and it would be a natural next step to incorporate this into our Lean formalization.

3. A concrete definition of polynomial time. While the axiomatic approach to formalizing polynomial time is clean and simple, it places the burden on the reader to verify that the axioms (including specific reductions) are valid. A more robust alternative would be to adopt a concrete definition of polynomial time. The calculi proposed in [6] and [28] appear promising in this regard. However, applying them in our framework seems very challenging and would probably outweigh its other parts. It is, however, possible that further developments in the Lean (or other proof assistants) ecosystem will make this easier in the future.

REFERENCES

[1] Martín Abadi and Phillip Rogaway. 2007. Reconciling Two Views of Cryptography (The Computational Soundness of Formal Encryption). *J. Cryptology* 20, 3 (2007), 395.

[2] Martín Abadi and Bogdan Warinschi. 2008. Security analysis of cryptographically controlled access to XML documents. *J. ACM* 55, 2 (2008), 6:1–6:29. https://doi.org/10.1145/1346330.1346331

[3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Guillaume Davy, François Dupressoir, Benjamin Grégoire, and Pierre-Yves Strub. 2014. Verified Implementations for Secure and Verifiable Computation. *IACR Cryptol. ePrint Arch.* (2014), 456. http://eprint.iacr.org/2014/456

[4] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, and Vitor Pereira. 2017. A Fast and Verified Software Stack for Secure Function Evaluation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 1989–2006. https://doi.org/10.1145/3133956.3134017

[5] José Bacelar Almeida, Manuel Barbosa, Manuel L. Correia, Karim Eldefrawy, Stéphane Graham-Lengrand, Hugo Pacheco, and Vitor Pereira. 2021. Machine-checked ZKP for NP relations: Formally Verified Security Proofs and Implementations of MPC-in-the-Head. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 2587–2600. https://doi.org/10.1145/3460120.3484771

[6] Robert Atkey. 2024. Polynomial Time and Dependent Types. *Proc. ACM Program. Lang.* 8, POPL (2024), 2288–2317. https://doi.org/10.1145/3632918

[7] Bolton Bailey and Andrew Miller. 2024. Formalizing Soundness Proofs of Linear PCP SNARKs. In *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*, Davide Balzarotti and Wenyuan Xu (Eds.). USENIX Association. https://www.usenix.org/conference/usenixsecurity24/presentation/bailey

[8] Gilles Barthe, Juan Manuel Crespo, Benjamin Grégoire, César Kunz, and Santiago Zanella-Béguelin. 2012. Computer-Aided Cryptographic Proofs. In *Interactive Theorem Proving - Third International Conference, ITP 2012, Princeton, NJ,*

USA, August 13-15, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7406), Lennart Beringer and Amy P. Felty (Eds.). Springer, 11–27. https://doi.org/10.1007/978-3-642-32347-8_2

[9] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. 2011. Computer-Aided Security Proofs for the Working Cryptographer. In *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*. 71–90. https://doi.org/10.1007/978-3-642-22792-9_5

[10] Mathieu Baudet, Bogdan Warinschi, and Martín Abadi. 2010. Guessing attacks and the computational soundness of static equivalence. *Journal of Computer Security* 18, 5 (2010), 909–968. https://doi.org/10.3233/JCS-2009-0386

[11] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. 2012. Foundations of garbled circuits. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, Ting Yu, George Danezis, and Virgil D. Gligor (Eds.). ACM, 784–796. https://doi.org/10.1145/2382196.2382279

[12] Nir Bitansky, Ran Canetti, Alessandro Chiesa, Shafi Goldwasser, Huijia Lin, Aviad Rubinstein, and Eran Tromer. 2017. The Hunting of the SNARK. *J. Cryptol.* 30, 4 (2017), 989–1066. https://doi.org/10.1007/S00145-016-9241-9

[13] Bruno Blanchet. 2006. A Computationally Sound Mechanized Prover for Security Protocols. In *IEEE Symposium on Security and Privacy*. Oakland, California, 140–154.

[14] Bruno Blanchet. 2013. Automatic Verification of Security Protocols in the Symbolic Model: The Verifier ProVerif. In *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures (Lecture Notes in Computer Science, Vol. 8604)*, Alessandro Aldini, Javier López, and Fabio Martinelli (Eds.). Springer, 54–87. https://doi.org/10.1007/978-3-319-10082-1_3

[15] Markus de Medeiros, Muhammad Naveed, Tancrède Lepoint, Temesghen Kahsai, Tristan Ravitch, Stefan Zetzsche, Anjali Joshi, Joseph Tassarotti, Aws Albarghouthi, and Jean-Baptiste Tristan. 2024. Verified Foundations for Differential Privacy. *IACR Cryptol. ePrint Arch.* (2024), 2040. https://eprint.iacr.org/2024/2040

[16] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12699)*, André Platzer and Geoff Sutcliffe (Eds.). Springer, 625–635. https://doi.org/10.1007/978-3-030-79876-5_37

[17] Karim Eldefrawy and Vitor Pereira. 2019. A High-Assurance Evaluator for Machine-Checked Secure Multiparty Computation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM, 851–868. https://doi.org/10.1145/3319535.3354205

[18] David Evans, Vladimir Kolesnikov, and Mike Rosulek. 2018. A Pragmatic Introduction to Secure Multi-Party Computation. *Found. Trends Priv. Secur.* 2, 2-3 (2018), 70–246. https://doi.org/10.1561/3300000019

[19] Shimon Even, Oded Goldreich, and Abraham Lempel. 1985. A Randomized Protocol for Signing Contracts. *Commun. ACM* 28, 6 (1985), 637–647. https://doi.org/10.1145/3812.3818

[20] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*. 218–229. https://doi.org/10.1145/28395.28420

[21] Shafi Goldwasser and Silvio Micali. 1984. Probabilistic Encryption. *J. Comput. Syst. Sci.* 28, 2 (1984), 270–299. https://doi.org/10.1016/0022-0000(84)90070-9

[22] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. 1989. The Knowledge Complexity of Interactive Proof Systems. *SIAM J. Comput.* 18, 1 (1989), 186–208. https://doi.org/10.1137/0218012

[23] Helene Haagh, Aleksandr Karbyshev, Sabine Oechsner, Bas Spitters, and Pierre-Yves Strub. 2018. Computer-Aided Proofs for Multiparty Computation with Active Security. In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*. IEEE Computer Society, 119–131. https://doi.org/10.1109/CSF.2018.00016

[24] Paul Hudak, Simon L. Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph H. Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Richard B. Kieburtz, Rishiyur S. Nikhil, Will Partain, and John Peterson. 1992. Report on the Programming Language Haskell, A Non-strict, Purely Functional Language. *ACM SIGPLAN Notices* 27, 5 (1992), 1. https://doi.org/10.1145/130697.130699

[25] Vladimir Kolesnikov and Thomas Schneider. 2008. Improved Garbled Circuit: Free XOR Gates and Applications. In *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II - Track B: Logic, Semantics, and Theory of Programming & Track C: Security and Cryptography Foundations (Lecture Notes in Computer Science, Vol. 5126)*, Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz (Eds.). Springer, 486–498. https://doi.org/10.1007/978-3-540-70583-3_40

[26] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. [n.d.]. The OCaml system: Documentation and user's manual. *INRIA* 3 ([n. d.]), 42.

[27] Baiyu Li and Daniele Micciancio. 2018. Symbolic security of garbled circuits. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 147–161.

[28] Kevin Liao, Matthew A. Hammer, and Andrew Miller. 2019. ILC: a calculus for composable, computational cryptography. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 640–654. https://doi.org/10.1145/3314221.3314607

[29] Yehuda Lindell and Benny Pinkas. 2009. A Proof of Security of Yao's Protocol for Two-Party Computation. *J. Cryptology* 22, 2 (2009), 161–188. https://doi.org/10.1007/s00145-008-9036-8

[30] Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. 2013. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8044)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, 696–701. https://doi.org/10.1007/978-3-642-39799-8_48

[31] Daniele Micciancio. 2010. Computational Soundness, Co-induction, and Encryption Cycles. In *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30 - June 3, 2010. Proceedings*. 362–380.

[32] Daniele Micciancio and Saurabh Panjwani. 2006. Corrupting One vs. Corrupting Many: The Case of Broadcast and Multicast Encryption. In *Automata, Languages and Programming, 33rd International Colloquium, ICALP 2006, Venice, Italy, July 10-14, 2006, Proceedings, Part II*. 70–82. https://doi.org/10.1007/11787006_7

[33] Daniele Micciancio and Saurabh Panjwani. 2008. Optimal communication complexity of generic multicast key distribution. *IEEE/ACM Trans. Netw.* 16, 4 (2008), 803–813. https://doi.org/10.1145/1453698.1453703

[34] Moni Naor and Benny Pinkas. 2001. Efficient oblivious transfer protocols. In *Proceedings of the Twelfth Annual Symposium on Discrete Algorithms, January 7-9, 2001, Washington, DC, USA*, S. Rao Kosaraju (Ed.). ACM/SIAM, 448–457. http://dl.acm.org/citation.cfm?id=365411.365502

[35] Saurabh Panjwani. 2007. Tackling Adaptive Corruptions in Multicast Encryption Protocols. In *Theory of Cryptography, 4th Theory of Cryptography Conference, TCC 2007, Amsterdam, The Netherlands, February 21-24, 2007, Proceedings*. 21–40.

[36] Chris Peikert, Vinod Vaikuntanathan, and Brent Waters. 2008. A Framework for Efficient and Composable Oblivious Transfer. In *Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 5157)*, David A. Wagner (Ed.). Springer, 554–571. https://doi.org/10.1007/978-3-540-85174-5_31

[37] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. 2009. Secure Two-Party Computation Is Practical. In *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5912)*, Mitsuru Matsui (Ed.). Springer, 250–267. https://doi.org/10.1007/978-3-642-10366-7_15

[38] Michael O. Rabin. 2005. How To Exchange Secrets with Oblivious Transfer. *IACR Cryptol. ePrint Arch.* (2005), 187. http://eprint.iacr.org/2005/187

[39] Wouter Swierstra. 2008. Data types à la carte. *J. Funct. Program.* 18, 4 (2008), 423–436. https://doi.org/10.1017/S0956796808006758

[40] Devon Tuma and Nicholas Hopper. 2024. VCVio: A Formally Verified Forking Lemma and Fiat-Shamir Transform, via a Flexible and Expressive Oracle Representation. *IACR Cryptol. ePrint Arch.* (2024), 1819. https://eprint.iacr.org/2024/1819

[41] Kaiyu Yang, Aidan M. Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J. Prenger, and Animashree Anandkumar. 2023. LeanDojo: Theorem Proving with Retrieval-Augmented Language Models. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (Eds.). http://papers.nips.cc/paper_files/paper/2023/hash/4441469427094f8873d0fecb0c4e1cee-Abstract-Datasets_and_Benchmarks.html

[42] Andrew Chi-Chih Yao. 1982. Protocols for Secure Computations (Extended Abstract). In *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*. 160–164.