ProofFrog: A Tool For Verifying Game-Hopping Proofs

Ross Evans
University of Waterloo
Waterloo, Canada
rpevans@uwaterloo.ca

Matthew McKague
Queensland University of Technology
Brisbane, Australia
matthew.mckague@qut.edu.au

Douglas Stebila
University of Waterloo
Waterloo, Canada
dstebila@uwaterloo.ca

***Abstract*—Cryptographic proofs allow researchers to provide theoretical guarantees on the security that their constructions provide. A proof of security can completely eliminate a class of attacks by potential adversaries. Human fallibility, however, means that even a proof reviewed by experts may still hide flaws or outright errors. Proof assistants are software tools built for the purpose of formally verifying each step in a proof, and as such have the potential to prevent erroneous proofs from being published and insecure constructions from being implemented. Unfortunately, existing tooling for verifying cryptographic proofs has found limited adoption in the cryptographic community, in part due to concerns with ease of use. We present ProofFrog: a new tool for verifying cryptographic game-hopping proofs. ProofFrog is designed with the average cryptographer in mind, using an imperative syntax similar to C for specifying games and a syntax for proofs that closely models pen-and-paper arguments. As opposed to other proof assistant tools which largely operate by manipulating logical formulae, ProofFrog manipulates abstract syntax trees (ASTs) into a canonical form to establish indistinguishable or equivalent behaviour for pairs of games in a user-provided sequence. We also detail the domain-specific language developed for use with the ProofFrog proof engine, the exact transformations it applies to canonicalize ASTs, and case studies of verified proofs. A tool like ProofFrog that prioritizes ease of use can lower the barrier of entry to using computer-verified proofs and aid in catching insecure constructions before they are made public.**

***Index Terms*—cryptography, game-hopping, proof verification**

I. Introduction

Computer-aided cryptography aims to increase confidence in the correctness of cryptographic constructions. The subject can be broadly categorized into three approaches: verifying implementation-level security, verifying functional correctness and efficiency, and verifying design-level security [1]. Design-level security tooling can then be further divided into those which work in the computational model versus those which work in the symbolic model. Tools such as ProVerif [2] and Tamarin [3] operate in the symbolic model where theorems are proved assuming atomic data and black-box cryptographic primitives; such proofs can be more amenable to verification but also provide weaker security guarantees due to the idealized model. In contrast, the *computational model*, which is the focus of this paper, can be less idealized as the underlying representation of data is considered and primitives are modelled as explicit probabilistic algorithms.

One technique for proving theorems in the computational model is *game hopping* [4], [5]. Game hopping proofs define security properties via games played between a challenger and an adversary, where the security property is satisfied so long as no probabilistic polynomial-time adversary can achieve a win condition with a non-negligible probability. A game-hopping proof uses a sequence of games and "hops" from one game to the next by making small changes to the game's definition, where each change alters the output distribution by a negligible (possibly zero) amount. This technique can help bound the probability of an adversary winning the initial game by gradually changing the game into an "unwinnable" game or one in which the adversary's probability of success is easy to calculate.

Tools for game hopping-proofs. There are a variety of tools that exist for the verification of game-hopping proofs. We now review two of the more prominent (EasyCrypt [6] and CryptoVerif [7]) as well as others.

EasyCrypt [6] utilizes an imperative language for specifying games and a formula language for specifying probabilistic relational Hoare logic judgments which define security properties. Actually writing a proof in EasyCrypt requires the use of their "tactic" language which allows one to manipulate game syntax and apply logical rules to manipulate the formula being proved. EasyCrypt itself is very expressive; it allows a user to prove very general statements, but at the cost of complexity. The tactic language applies manipulations at a fine level of detail, where even simple axioms, such as associativity of group operations, must be explicitly applied each time they are used. In addition, EasyCrypt proofs can also be difficult to read, since it can be unclear what formula each tactic is being applied to unless one steps through the proof interactively in an editor.

CryptoVerif [7] leans in the opposite direction: rather than focusing on expressiveness, it focuses on automation. It allows users to specify games via a process calculus syntax akin to that of functional programming languages. Proofs can often be discovered automatically by the proof engine, but it also supports an interactive mode where a user can explicitly specify which game transformations they want applied if the proof engine fails to automatically discover a proof. Similarly to EasyCrypt, security properties are also expressed as logical

formulae, except in CryptoVerif these formulae are internal rather than user-defined, and are verified by an internal equational prover rather than manually by the user.

In addition to the tools previously mentioned, a number of other tools exist in a variety of languages for different frameworks. EasyUC [8] formalizes universal composability — an alternative model outside of the game-based approach — within EasyCrypt. SSProve [9] formalizes state-separating proofs — which attempt to structure game-based proofs more modularly — within Coq. Squirrel [10] develops a higher-order logic based on the computationally complete symbolic attacker framework. CryptHol [11] formalizes cryptographic arguments in Isabelle/HOL. Finally, some efforts have been made to formalize protocols such as TLS in F* [12], which is a language that works both as a proof assistant as well as a general-purpose programming language.

Contributions. This paper introduces ProofFrog[1]: a new tool for verifying cryptographic game-hopping proofs. ProofFrog takes a novel approach in that it focuses purely on high-level manipulations of games as abstract syntax trees (ASTs) instead of working at the level of logical formulae. Treating games as ASTs allows us to leverage techniques from compiler design and static analysis to prove output equivalence of games, thereby allowing us to demonstrate the validity of hops in a game sequence. The main technique used in our engine is to take pairs of game ASTs and perform a variety of transformations in an attempt to coerce the two ASTs into canonical forms, which can then be compared. If the canonical forms are identical then the two games are equivalent and our proof engine can assert the validity of the hop. These transformations are performed to the AST with little user guidance, which makes writing a proof in many cases as simple as just specifying which reductions are being leveraged. This approach also extends to hybrid arguments [13] which rely on a bounded induction through a sequence of related reductions. ProofFrog verifies hybrid arguments by ensuring equivalence of ASTs for each hop within the induction and for the boundary conditions at both ends.

ProofFrog also targets ease of use: although it implements a domain-specific language that a user must learn, the language has an imperative C-like syntax that should be comfortable for the average cryptographer. The proof syntax is intentionally designed for improved readability by closely mimicking that of a typical pen-and-paper proof. Throughout ProofFrog's development we assembled a corpus of proofs from *The Joy of Cryptography* [14] which were easily translated into ProofFrog's syntax with minimal changes. ProofFrog also supports type checking functionality to reject ill-formed reductions with clear explanations of why they are invalid. Beyond its use as a general proof verification tool, we foresee ProofFrog being useful as an educational tool for beginners learning cryptographic proofs.

[1]The implementation of ProofFrog is available at https://github.com/ProofFrog/ProofFrog, example files are available at https://github.com/ProofFrog/examples

The approach of canonicalizing ASTs does have some limitations; it can be difficult for ProofFrog to reason about long-term state in games, resulting in potential simplifications being missed, and limiting the class of games and schemes which ProofFrog can successfully reason about. In addition, our work does not provide any formal proofs of correctness for the transformations ProofFrog uses or for the correctness of the engine's implementation. ProofFrog is not alone in requiring user trust however, previously mentioned tools such as EasyCrypt and CryptoVerif also rely on their own trusted computing base.

The rest of the paper proceeds as follows. Section II discusses necessary background. In Section III and Figure 1, we provide a high-level overview of ProofFrog's approach, accompanied by a worked example proof in Listings 1 to 3. In Section IV we discuss the preprocessing and postprocessing steps ProofFrog uses on every proof it parses. Section V provides a detailed view into ProofFrog's strategies for canonicalizing abstract syntax trees. Section VI elaborates on other features included as part of ProofFrog. Finally, Section VII presents case studies and Section VIII concludes with discussion and avenues for future work.

II. Background

Like [5], [14], we consider games to be packages of code that provide oracle methods to an adversary $\mathcal{A}$, which is simply a program that can call the provided oracle methods and outputs a bit $b \in \{0, 1\}$. A pair of games G_L, G_R given in a security definition will differ in their behaviour, and the adversary's goal is to determine whether they are interacting with G_L or G_R. We use the notation $G \circ \mathcal{A}$ to denote the composition of an adversary with a game, where composition simply means using the code of G's oracles to answer $\mathcal{A}$'s queries. In such a case, we call G the "challenger" to the adversary $\mathcal{A}$. We use the notation $\Pr[G \circ \mathcal{A} \to b']$ to denote the probability that adversary $\mathcal{A}$ outputs bit b' when given access to G's oracles.

Definition 1: The **distinguishing advantage** of an adversary $\mathcal{A}$ for two games G_L and G_R is defined as:

$$\mathrm{Adv}(\mathcal{A}, G_L, G_R) = |\Pr[G_L \circ \mathcal{A} \to 1] - \Pr[G_R \circ \mathcal{A} \to 1]|$$

Definition 2: Two games G_L and G_R are **interchangeable** (denoted $G_L \equiv G_R$) if and only if for every adversary $\mathcal{A}$:

$$\mathrm{Adv}(\mathcal{A}, G_L, G_R) = 0$$

Definition 3: A function $f(\lambda)$ is **negligible** if for every polynomial p there exists an N such that for all $\lambda > N$, $f(\lambda) < \frac{1}{p(\lambda)}$

Definition 4: Two games G_L and G_R are **indistinguishable** (with notation $G_L \approx G_R$) if and only if for all probabilistic polynomial-time adversaries $\mathcal{A}$, $\mathrm{Adv}(\mathcal{A}, G_L, G_R)$ is negligible. Both the running time of $\mathcal{A}$ and the distinguishing advantage are calculated with respect to a **security parameter**

λ which is provided as input to the adversary $\mathcal{A}$ and the games G_L and G_R in unary.[2]

Note that although we only consider security definitions encoded as pairs of indistinguishable games, this restriction does not exclude security properties typically defined as win-lose games, such as unforgeability of a signature scheme or message authentication code. Any security definition predicated on achieving some win condition can be easily translated into a pair of games which are identical apart from a differing `CheckWin` oracle. One game's `CheckWin` oracle will actually perform the computation to check the win condition whereas the other will always return false; the adversary can distinguish between this pair of games if and only if they can achieve the win condition.

III. High Level Behaviour

To utilize ProofFrog, a user implements their security definitions, cryptographic constructions, and proofs in the ProofFrog domain-specific language, which has a C- or Java-like syntax (see Appendix C). ProofFrog parses these inputs, and processes them as shown in in Fig. 1 to assess the validity of the proof.

In this section, we walk through an example of this domain specific language encoding a proof that a symmetric encryption scheme satisfying CPA$ security (ciphertexts are indistinguishable from random, under a chosen plaintext attack) is also CPA-secure (semantic security (a.k.a., indistinguishability of ciphertexts) under a chosen plaintext attack).

Defining the primitive. Listing 1 shows the primitive `SymEnc` representing a symmetric encryption scheme. The `SymEnc` primitive defines internal types `Key`, `Message`, and `Ciphertext` based on the parameters the SymEnc primitive is instantiated with, and provides interfaces for the `KeyGen`, `Enc` and `Dec` methods.

Defining the security property. Listing 2 shows pairs of games for the CPA$ and CPA security properties. The `CPA` security definition defines games named `Left` and `Right`, while the `CPA$` security definition defines games named `Real` and `Random`. The `CPA` games each initialize a long-term key `k` from their provided `SymEnc` scheme, and provide an oracle `Eavesdrop` which takes two messages `mL` and `mR` and returns the encryption of `mL` with `k` in the `Left` game and the encryption of `mR` with `k` in the `Right` game. The `Real` `CPA$` game initializes a long-term key `k` from its provided `SymEnc` scheme and provides an oracle `CTXT` which takes a message `m` and returns the encryption of `m` with `k`, whereas the `Random` game omits any initialization, instead simply returning a random ciphertext whenever its `CTXT` oracle is called.

Stating the theorem and proof. Listing 3 contains a reductionist proof which demonstrates that a symmetric encryption scheme which satisfies the CPA$ security property also satisfies the CPA security property. The proof file lists two reductions that are used in the sequence of games. A proof consists of a `let` section, which details the variables, primitives, and schemes to be used in the proof, an `assume` section, which details what indistinguishability assumptions will be used in the proof, a `theorem` section, which states which security definition is to be proved, and a `proof` section. The `proof` section lists a sequence of steps, starting with one game in the security definition and ending with the other. Each step consists of either a game, a reduction composed with a game, or a block of games used in an inductive argument.

Checking a proof. ProofFrog will iterate through the steps of the proof and check that each step is either indistinguishable or interchangeable with its neighbours. If all these checks pass then ProofFrog accepts, otherwise it rejects and indicates which step could not be verified.

To verify indistinguishability between pairs of games, ProofFrog will simply check that the two games are listed as indistinguishable by assumption in the proof file. Otherwise, ProofFrog will verify that the two games are interchangeable. To do so, it attempts to coerce each game into a canonical form, such that interchangeability can be established via syntactic equivalence. Coercing each game into a canonical form typically requires composing a game and a reduction into a single game, then applying transformations to the ASTs so that ASTs can be appropriately compared. In addition, ProofFrog will rewrite its parameters in terms of those defined in the proof.

In Section IV, we will explain preprocessing and post-processing steps applied to the games in a proof to prepare them for canonicalization; we also give specific examples with reference to the proof given in Listing 3. Then in Section V, we examine the transformations used to coerce an abstract syntax tree into a canonical form.

IV. Preprocessing and Postprocessing Steps

When verifying a proof, ProofFrog will attempt to demonstrate that hops without reductions are interchangeable and that hops with reductions are indistinguishable by some listed assumption. For the CPA$ $\implies$ CPA example of Listing 3, ProofFrog's objective is to verify that the hops from step 1 to step 2, step 3 to step 4, and step 5 to step 6 are all between interchangeable games, and that the hops from step 2 to step 3 and from step 4 to step 5 are indistinguishable by assumption. Typically, verifying interchangeability will require at least some canonicalizing transformations, however this proof is simple enough to be verified simply using only the engine's preprocessing and postprocessing steps.

A. Verifying Indistinguishability

ProofFrog judges two steps in a hop listed in the `games` section of the proof file to be *indistinguishable* if all of the following conditions are satisfied:

1) Both steps in the hop use a reduction.
2) The steps are identical apart from changing which game the reduction is composed with.

[2]ProofFrog is actually agnostic to whether security notions are modeled asymptotically (polynomial-time adversaries with negligible success probability) or concretely (t-time adversarties with ϵ success probability), and simplyly accumulates the advantage loss incurred with each hop of the proof.

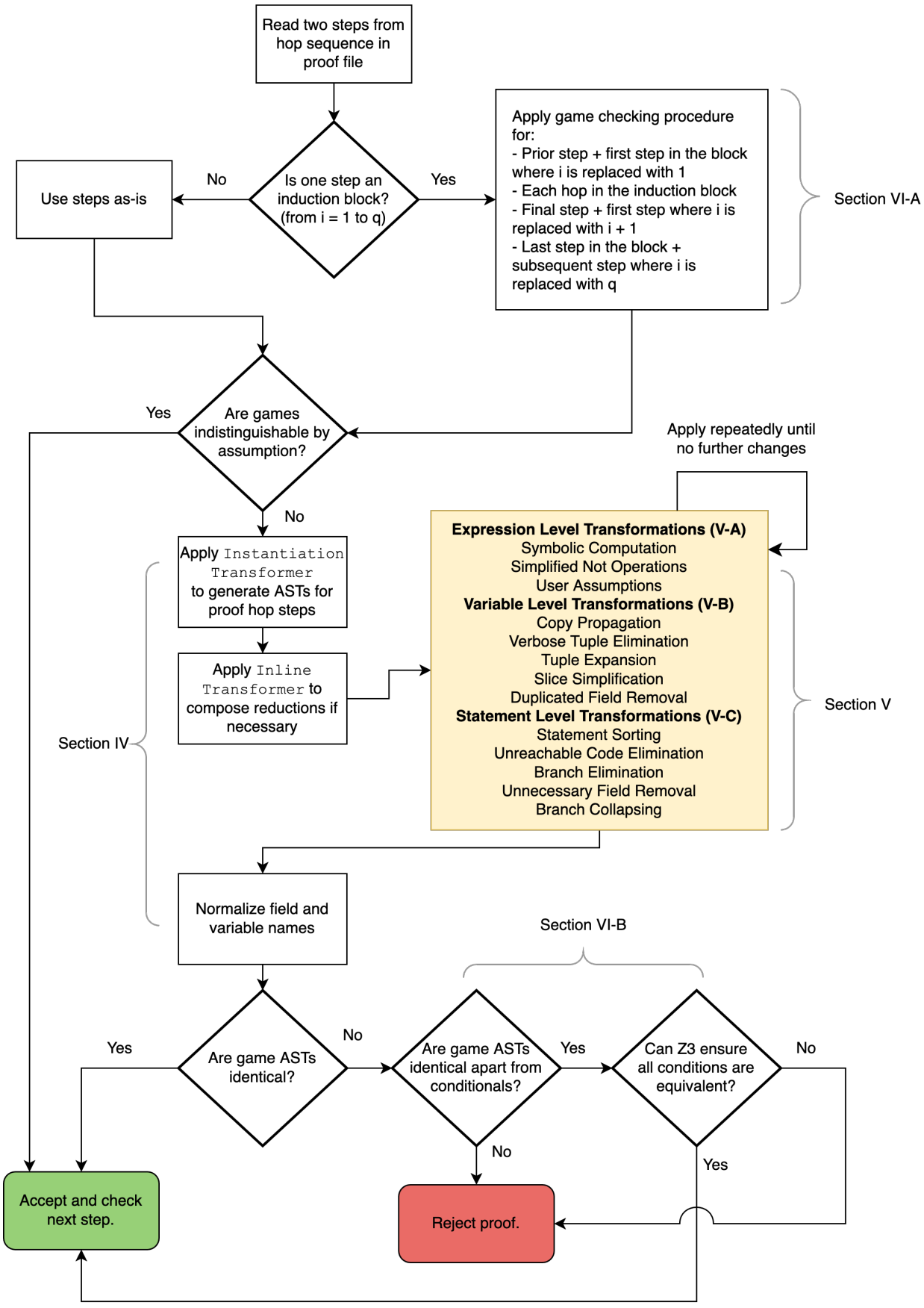

Fig. 1. ProofFrog's algorithm for checking indistinguishability or interchangeability of two hops in a game-hopping proof

```
Primitive SymEnc(Set M, Set C, Set K) {
  Set Message = M;
  Set Ciphertext = C;
  Set Key = K;
  Key KeyGen();
  Ciphertext Enc(Key k, Message m);
  Message Dec(Key k, Ciphertext c);
}
```

Listing 1: ProofFrog definition of `SymEnc` primitive

```
Game Real(SymEnc E) {
  E.Key k;
  Void Initialize() {
    k = E.KeyGen();
  }
  E.Ciphertext CTXT(E.Message m) {
    return E.Enc(k, m);
  }
}
Game Random(SymEnc E) {
  E.Key k;
  Void Initialize() {
    k = E.KeyGen();
  }
  E.Ciphertext CTXT(E.Message m) {
    E.Ciphertext c <- E.Ciphertext;
    return c;
  }
}
export as CPA$;
```

```
Game Left(SymEnc E) {
  E.Key k;
  Void Initialize() {
    k = E.KeyGen();
  }
  E.Ciphertext Eavesdrop(E.Message mL, E.Message mR) {
    return E.Enc(k, mL);
  }
}
Game Right(SymEnc E) {
  E.Key k;
  Void Initialize() {
    k = E.KeyGen();
  }
  E.Ciphertext Eavesdrop(E.Message mL, E.Message mR) {
    return E.Enc(k, mR);
  }
}
export as CPA;
```

Listing 2: ProofFrog definitions of `CPA$` (left) and `CPA` (right) security games for symmetric encryption

```
Reduction R1(SymEnc E1) compose CPA$(E1) against CPA(E1).Adversary {
  E1.Ciphertext Eavesdrop(E1.Message mL, E1.Message mR) {
    return challenger.CTXT(mL);
  }
}
Reduction R2(SymEnc E2) compose CPA$(E2) against CPA(E2).Adversary {
  E2.Ciphertext Eavesdrop(E2.Message mL, E2.Message mR) {
    return challenger.CTXT(mR);
  }
}
proof:
let:
  Set MessageSpace;
  Set CiphertextSpace;
  Set KeySpace;
  SymEnc E = SymEnc(MessageSpace, CiphertextSpace, KeySpace);
assume:
  CPA$(E);
theorem:
  CPA(E);
games:
  CPA(E).Left against CPA(E).Adversary;                  // Step 1
  CPA$(E).Real compose R1(E) against CPA(E).Adversary;   // Step 2
  CPA$(E).Random compose R1(E) against CPA(E).Adversary; // Step 3
  CPA$(E).Random compose R2(E) against CPA(E).Adversary; // Step 4
  CPA$(E).Real compose R2(E) against CPA(E).Adversary;   // Step 5
  CPA(E).Right against CPA(E).Adversary;                 // Step 6
```

Listing 3: ProofFrog proof that a CPA$-secure symmetric encryption scheme is CPA-secure

3) The two games the reduction is composed with in the two steps are the pair of games in a security definition.
4) This security definition appears in the `assume` section of the proof with identical parameters.

For example, the second hop in Listing 3, from `CPA$(E).Real compose R1(E)` to `CPA$(E).Random compose R1(E)` satisfies all four of these criteria, and hence is a valid hop by *indistinguishability*.

On the other hand, if anything else changes in the step, e.g, the reduction being used, the reduction's parameters, the challenger's parameters, etc., or, if the security definition does not appear in the `assumptions` section, then this step cannot be assumed by indistinguishability and must be checked for *interchangeability*. For example, the hop from `CPA$(E).Random compose R1(E)` to `CPA$(E).Random compose R2(E)` must be checked for interchangeability because the reduction being applied is not the same in each step.

B. Verifying Interchangeability

To validate interchangeability, ProofFrog transforms steps involving reductions into single games, so that the AST can be simplified into a canonical form and then compared. Validating a hop like `CPA(S).Left` to `CPA$(E).Real compose R1(E)` (step 1 to step 2 in Listing 3) requires taking the definitions of `CPA$(E).Real` and `R1(E)` and composing them into a single game.

Before performing composition, the `InstantiationTransformer` will create copies of definitions that are parameterized and replace references to these parameters with values that are defined in the proof's `let` section. As an example, consider the definition `SymEnc E = SymEnc(MessageSpace, CiphertextSpace, KeySpace)`. ProofFrog will associate with the value `E` a copy of the `SymEnc` primitive AST, where any internal references to the parameters in the `SymEnc` definition (i.e. `M`, `C`, `K`) are replaced with `MessageSpace`, `CiphertextSpace`, and `KeySpace` respectively. The `InstantiationTransformer` is also applied for game and reduction definitions: the value `CPA(E).Left` is associated with the CPA left game AST. This AST may also refer to type fields of `E` such as `E.Key` which are rewritten with their associated definitions in the `let` section. When all references to parameters have been rewritten in terms of variables defined in the `let` section, these definition copies have their parameters removed. The `InstantiationTransformer` helps ensure consistency in parameterized variables across ASTs: for example, two games may both be parameterized with a `SymEnc` parameter named `E`, but if the steps instantiate the games with two different schemes, then it would be a mistake to compare the ASTs as equal just because the parameter name is the same.

Creating the Inlined Game. The `InstantiationTransformer` alone is enough to prepare a game AST representing `CPA(E).Left`. The task of composing `CPA$(E).Real` and `R1(E)` into a single game, which we call the *inlined game*, remains. To do so, we use the AST associated with `R1(E)` as a base to modify, since `R1(E)` already defines the oracles that the adversary would expect when interacting with a single game.

The first step is to combine states together: each field included in the AST associated with `CPA$(E).Real` and the AST associated with `R1(E)` should be included in the inlined game. Variable renaming is undertaken to ensure no conflicts: if both the reduction and the challenger have a field `k`, it could be ambiguous which field is being referred to in the inlined game. To avoid conflicts, each field `f` from the challenger is renamed to `challenger@f`, and any references to the field in the challenger AST are rewritten accordingly before composition. The `@` symbol is not permitted in variable names, so these names are guaranteed to be conflict-free.

The second step is to combine `Initialize` methods together. Games have `Initialize` methods with empty parameters and the option to return a value which is given to the adversary. A reduction can access this state from the challenger by accepting a parameter in its `Initialize` method; For the inlined game to compare identically to a regular game, ProofFrog must ensure its `Initialize` method is parameterless. When combining the challenger's and the reduction's `Initialize` methods together, ProofFrog will automatically insert a call to `challenger.Initialize()` as the first statement in the reduction's `Initialize` method. If the reduction's `Initialize` method takes in a parameter, ProofFrog will also create a local variable to capture the return value of `challenger.Initialize()` with the same name as the parameter, which ensures that the remaining statements function as expected. This transformation to the reduction's `Initialize` method ensures that when ProofFrog begins inlining challenger method calls, the inlined game will consist of an `Initialize` method with no parameters, and method signatures that match those expected by the adversary.

Finally, to actually create the inlined game AST, ProofFrog must remove any calls to challenger oracles inside the reduction. ProofFrog uses a method inlining strategy to achieve this. The `InlineTransformer` searches each method in the reduction to find the first function call expression to a challenger oracle. It then uses the `InstantiationTransformer` to create a copy of the challenger oracle's AST where the parameters have been replaced with the arguments provided to the call. Additionally, the `InlineTransformer` performs renaming of local variables in the challenger oracle's AST by prefixing them with the oracle name and the `@` symbol, so as to avoid conflicts when inlining the code into the reduction's method. If the reduction calls the oracle solely for its side effects, then the transformation is completed by replacing the function call with the statements from the modified oracle AST. On the other hand, if the result of the challenger oracle call is saved to a local variable, then the `InlineTransformer` will replace the challenger function call expression in the reduction's method with the expression found in the return statement of the oracle body, and then remove the associated return statement from the

oracle before inserting the prior statements into the reduction. For simplicity, ProofFrog assumes that return statements are only placed as the final statement for any challenger oracles. The `InlineTransformer` repeats this process of inlining oracle calls until there are none left in any methods of the inlined game AST.

After completing each of these steps—combining the states of the challenger and the reduction, combining the `Initialize` methods, and inlining any challenger calls—the inlined game AST is now a complete representation of the behaviour of `CPA$(E).Real compose R1(E)`, written as a single game. From here, canonicalizing transformations would typically be applied to simplify the AST representations of both `CPA(E).Left` and `CPA$(E).Real compose R1(E)`. However, none of the steps in this proof require any canonicalizing transformations.

We now discuss the final postprocessing step necessary for verifying all steps in this proof.

Normalizing Field and Variable Names. The inlined game AST and the AST for `CPA(E).Left` do not compare as identical after preprocessing, because variable names are mismatched. The inlined game uses the variable name `challenger@k` for its long-term key whereas `CPA(E).Left` simply uses `k`. To address this issue, ProofFrog normalizes the names of the fields for all games and the names of the variables for all oracles contained within each game. Each field's name is converted to `fieldx`, where `x` is an index representing the order of occurrence when traversing the oracles of the game AST. Variables are renamed similarly. This canonicalization of variables is enough for `CPA(E).Left` and the inlined game AST to match identically. These preprocessing and postprocessing actions are enough to validate all interchangeability steps in this proof.

V. Transformations

While the preprocessing and postprocessing actions previously described were enough to verify the proof in Listing 3, more complex proofs may require more additional simplification techniques to assess interchangeability of games. This section will detail the various transformations used by ProofFrog to validate interchangeability of games[3]. For explanatory purposes, we have grouped these transformations into three broad categories: *expression level* transformations, which rewrite expressions in simpler forms, *variable level* transformations, which concern the manipulation of variables with multiple representations into a single form, and *statement level* transformations, which concern the ordering, simplification, and removal of statements or blocks within the program.

A. Expression Level Transformations

Symbolic Computation. SymPy — a library developed for symbolic computation and computer algebra in Python [15] — is used in ProofFrog for simplifying mathematical expressions. For an expression to be eligible, it must be an addition, subtraction, multiplication, or division operation applied to two values. These values must either be integers, or variables with an integer type. For each variable encountered with an integer type, ProofFrog will create a corresponding SymPy symbol. Then, when an expression like `lambda + lambda` is encountered, the same SymPy symbol is produced for both variables, which allows SymPy to simplify to a value like `2 * lambda`. Variables with differing names will produce differing symbols which SymPy will not collapse. If an eligible expression can be simplified, we then convert the SymPy AST into the equivalent ProofFrog AST, and replace the expression node with a simplified node. This transformation is mainly used in ProofFrog to canonicalize the length parameterizations for `BitString`s, but can also simplify numerical expressions when relevant.

Simplified Not Operations. Consider an oracle `f` which returns some equality check `A == B`. A reduction may wish to use the negation of this equality check by calling `!f()`. Inlining yields the expression `!(A == B)`, which is canonicalized to `A != B`.

User Assumptions. In addition to its automatic transformations, ProofFrog allows users to specify assumptions between hops that can aid the engine in canonicalization. These assumptions take the form of boolean statements about game variables. If a particular expression in the game is identical to a provided assumption, then the value can be directly replaced with `true`. Otherwise, ProofFrog will delegate to Z3, a satisfiability modulo theories (SMT) solver used extensively for static analysis of programs which can in many cases determine whether logical formulae are satisfiable or unsatisfiable [16]. The user assumption will be converted to a Z3 formula to determine if any expressions are provably true or false under this assumption.

When evaluating a formula, Z3 will produce one of three results: a satisfying assignment to the variables, a certificate that the formula is unsatisfiable, or unknown. There are two cases in which we can simplify an expression e by using an assumption a. First, if $a \implies e$ is a tautology, then since a is assumed to be true, we must have that e is true as well. Second, if $a \wedge e$ is a contradiction (unsatisfiable, in Z3 parlance) then since a is assumed to be true, we must have that e is false. Since Z3 does not have the ability to evaluate tautologies, we can instead use Z3 to evaluate $\neg(a \implies e)$. If this formula is unsatisfiable, we can directly replace e with `true`. Otherwise, we can evaluate $a \wedge e$, and replace e with `false` if Z3 deems the formula unsatisfiable. The use of Z3 allows us to replace some conditionals directly with boolean values which can aid in further simplification of games.

Allowing user-specification of assumptions does add some risk, as it is possible for a user to specify a false assumption and cause ProofFrog to come to an invalid conclusion. Any proofs that utilize user-specified assumptions would require a reader to check each assumption used and come to their own conclusion about the assumption's correctness.

[3]For an alternative presentation where transformations are discussed in the context of the proofs they were developed for, see the accompanying thesis.

B. Variable Level Transformations

Copy Propagation. Copy propagation is a technique used in compiler optimizations to reduce redundant computations. Some compiler transformations can introduce variables which are direct copies of others; copy propagation acts to remove such direct copies, replacing them with the original definition where possible to improve run-time performance [17, Chapter 9.1.5]. A few of ProofFrog's transformations similarly introduce copies of variables, such as method inlining. If a method `g` returns a variable `x` and another method `f` contains the statement `y = g();`, then inlining will result in the statements of `g` followed by the statement `y = x;`. The variable `y` is redundant as it will contain the same value as `x` for its entire lifetime. The copy propagation technique is repurposed in ProofFrog for canonicalizing ASTs instead of improving run-time performance. The engine searches in code blocks for direct copies: those which define a new variable (say, `b`) from an already existing variable (say, `a`) in the same scope. If the original variable `a` is never again used for the duration of its scope, then `b` "took over" the value of `a` from that point onwards. As a result, ProofFrog can remove the assignment to the variable `b` and rename any of its usages to `a`. This transformation preserves the behaviour of the code, while removing unnecessary duplicated variables.

Verbose Tuple Elimination. ProofFrog supports tuple types to allow for multiple return values from methods or oracles. However, during the process of inlining, tuples may become "verbose", in that expressions may appear in the form

```
[t[0], t[1], ..., t[n]]
```

instead of simply `t`. Verbose tuples can appear when an oracle takes in multiple arguments and packages them into a tuple. If a reduction calling this oracle passes the individual values of a tuple, then inlining yields a verbose tuple expression. ProofFrog detects verbose tuple expressions where each indexed value is a constant integer and rewrites the expression to just use the tuple itself.

Tuple Expansion. This transformation takes tuples and rewrites them in terms of individual variables. For example, a tuple like `Int * Int t = [a, b]` would be rewritten into two individual statements: `Int t@0 = a;` and `Int t@1 = b;`. For a particular tuple to be eligible for expansion, it must satisfy two conditions:

1) Whenever an element of the tuple is read or written to, the index used must be a constant integer.
2) Whenever the tuple itself is assigned, the value must also be a tuple AST node.

If either of these are violated, the tuple will not be expanded by ProofFrog.

For each tuple that is considered eligible, the transformer will perform the following transformations:

1) Rewrite `t = [v0, v1, ..., vn]` into $n + 1$ statements: `t@0 = v0; t@1 = v1; ... t@n = vn;`.
2) Rewrite `t[i]` where `i` is a constant integer into `t@i`
3) Rewrite any usages of `t` itself into `[t@0, t@1, ..., t@n]`.

Each of these transformations will apply for the entirety of `t`'s scope. This yields a new AST that has identical behaviour to the original, except `t` has been removed.

Slice Simplification. ProofFrog provides a `BitString` type parameterized by its length, as well as concatenation and slice operations to manipulate `BitString`s. Some reductions result in situations like the following:

```
BitString<lambda> a <- BitString<lambda>;
BitString<lambda> b <- BitString<lambda>;
BitString<2 * lambda> r = a || b;
BitString<lambda> x = r[0 : lambda];
BitString<lambda> y = r[lambda : 2*lambda];
```

The values `x` and `y` are directly extracted from `r` and will maintain the same values as `a` and `b`. To handle such cases, ProofFrog searches for assignments to variables that are the concatenation of two bitstrings (say `r = a || b`). Then, the transformer searches subsequent statements in that block for a statement satisfying a few conditions:

- The statement must create a new variable (say `x`) as the result of a slice of `r`.
- The statement must slice out exactly one of the previously defined variables (either `a` or `b`) from `r`.
- No changes should have been made to the original variable (`a` or `b`) or the concatenated variable `r` between the creation of `r` and the the creation of `x`.

If all of these conditions are satisfied, then ProofFrog will instead assign `x` directly to the value that was sliced out of it, whether that is `a` or `b`.

Duplicated Field Removal. Inlining reductions may result in duplicated fields: for example, if a game generates a public key during initialization and returns this value to a reduction, then the inlined game will contain two fields storing the same public key. To detect duplicated fields, ProofFrog will begin under the assumption that all pairs of fields (`f1, f2`) with the same type are duplicates. It will then iterate through each block attempting to pair statements that modify either `f1` or `f2` with a subsequent statement assigning the other field the same value. Assume without loss of generality that ProofFrog encounters the statement `f1 = e;`, where `e` is some expression. ProofFrog will iterate through subsequent statements for one of two conditions:

1) `f2 = f1;`, where neither `f2` nor `f1` have been used in any intermediate statements between `f1 = e;` and `f2 = f1;`.
2) `f2 = e;`, where `e` does not contain a function call, neither `f2` nor `f1` have been used, and none of the variables in `e` have been modified in any intermediate statements between `f1 = e;` and `f2 = e;`.

If neither of these conditions can be satisfied, then `f1` and `f2` are deemed not duplicates, and the next pair is inspected. On the other hand, if these conditions are satisfied each time `f1`

or `f2` is modified, then `f1` and `f2` are duplicates, and each subsequent statement that was found is denoted as a "matched statement". ProofFrog then transforms the AST by removing `f2`'s definition in the game's list of fields, replacing all uses of `f2` with `f1`, and removing all matched statements from the AST.

C. Statement Level Transformations

Statement Sorting. Two ASTs may have identical outputs with differing statement orderings. For example, in the following code block, the order of the variable declarations can be swapped without changing the behaviour of the program.

```
x = 1;
y = 2;
return x + y;
```

Inlining games into reductions can result in situations like this, where the two ASTs have identical output but the act of writing a reduction forces a particular ordering of statements which differs from the game in the next hop. To handle such cases, ProofFrog canonicalizes ordering of statements. ProofFrog's strategy to ensure that interchangeable ASTs have identical statement orderings is achieved via creating a dependency graph for each block followed by a topological sorting of the statements. While the topological sort cannot guarantee a canonical ordering of statements for every block, it has proven effective for the suite of proofs it has been tested upon. ProofFrog will consider a statement `s` in a block to depend on a prior statement `t` if any of the following conditions are satisfied:

1) If `s` is a return statement or contains a return statement in a nested block and `t` is a return statement or contains a return statement in a nested block, then `s` depends on `t`.
2) If `s` is a return statement or contains a return statement in a nested block and `t` assigns to a field, then `s` depends on `t`.
3) If `s` references a variable `a` and `t` also references `a`, then `s` depends on `t`.

The first point is a dependence as reordering statements which contain `returns` (if-statements, for example), could result in a different return value if both statements evaluate to `true`. The second point is a dependence because returning before assigning to fields could alter the results of later oracle calls. Finally, the third point is a dependence as reordering statements could change the values of variables and hence the output of an oracle.

The dependency graph is then utilized in Algorithm 1 to sort the statements inside a block. First, a depth-first traversal of the dependency graph starting from the first non-nested `return` statement is used to create a list of statements. The neighbor nodes are traversed according to the order of appearance of variables in the current node's corresponding statement. Assuming that the two games contain an identical list of statements up to variable renaming, then the traversal will produce the same ordering of statements for both games. This step provides a well-defined ordering between statements in situations where dependencies do not, such as between the statements `x = 1;` and `y = 2;` in our short example above. We then use Kahn's algorithm to topologically sort the list of statements created by the traversal according to the dependency graph [18]. Our implementation of the loop in line 15 uses the statement ordering from the depth-first traversal, so if Khan's algorithm does not otherwise prefer one statement over another, they will be enqueued according to the depth-first traversal ordering. This provides additional constraints on the canonical ordering with a lower priority than the dependency graph. This approach yields a canonical ordering of statements for any block that contains a return statement.

Algorithm 1 Topological Sort

Require: The block's final statement is a return statement

```
1:  visited_statements = empty stack
2:  Generate dependency graph G for the block
3:  Perform depth first traversal according to G starting with
    the return statement. Push to visited_statements
    for each statement visited.
4:  kahn_queue = empty queue
5:  sorted_statements = empty list
6:  while visited_statements is not empty do
7:      Pop s from visited_statements
8:      if s has no dependencies in G then
9:          Enqueue statement to kahn_queue
10:     end if
11: end while
12: while kahn_queue is not empty do
13:     Dequeue s from kahn_queue
14:     Append s to sorted_statements
15:     for each edge (r, s) in G do
16:         Remove (r, s) in G
17:         if r has no out-edges then
18:             Enqueue r to kahn_queue
19:         end if
20:     end for
21: end while
22: return sorted_statements
```

There are some limitations with this approach; mainly, if the block does not end with a return statement, then there is no clear statement from which the traversal should originate. A block like `{ a = 1; b = 2; }`, which depends on variables declared outside of the block, can only be canonically reordered with further context that a block-level analysis cannot provide. Furthermore, it is possible that statements may differ while still being interchangeable: for example, a game that contains the statement `return a + b` will have a different traversal than one with the statement `return b + a`. Different traversals will result in different statement orderings which would prevent the ASTs from becoming identical, even though the behaviour of each statement is the same. Hence, statements still require canonicalization on an

individual level or else the sorting procedure is ineffective. Nevertheless, this sorting approach suffices for all proofs that were implemented as part of ProofFrog's test suite.

Unreachable Code Elimination. After performing inlining, it may be possible to determine that some code is unreachable. For example, in the following code block, if `challenger.f()` simplifies to the condition `x <= 0` then all subsequent statements are unreachable:

```
if (x > 0) {
    return 1;
}
if (challenger.f(x)) {
    return 2;
}
...
```

Determining that subsequent statements are unreachable requires reasoning about when the disjunction of if-statements with unconditional returns forms a tautology. For this task, ProofFrog also delegates to Z3; the algorithm is described in Algorithm 2.

The goal is to determine at which point in a block a return statement will definitely have been reached. Essentially, we build up a formula f which is a disjunction of all conditions so far that would have caused a return. If we ever encounter an if-statement with an else branch where all blocks have an unconditional return, then we know that all statements after this point are unreachable. Otherwise, we take the current if or else-if condition and convert it into a Z3 formula for use. If Z3 can reason directly about the types and operations inside the condition (for example, integers and operations over integers), then these are encoded directly in the Z3 formula. However, with ProofFrog allowing abstract user-defined types, there may be conditions using types and operations that Z3 cannot reason about. Nevertheless, since the condition in an if-statement is guaranteed to return a boolean, we can map the condition itself, in its entirety, to a boolean in Z3. Each condition can therefore be mapped to a Z3 formula, and if its corresponding if-statement contains an unconditional return, then we add it to f's ongoing disjunction. The list l is used to keep track of the prior conditions in an if/else-if statement: for the current condition c in a chain of if/else-if conditions to cause a return, we must have that c is `true` while all prior conditions were `false`.

Finally, the `var_version_map` assigns a version number to each variable, which our Z3 formula conversion uses when mapping values like `c in S` to Z3 booleans. If we encounter `c in S` followed by `!(c in S)`, then these should map into a Z3 boolean and the negation of that same Z3 boolean. On the other hand, if `c` or `S` have been assigned to in between these two expressions, then we cannot guarantee that the operation `c in S` would have the same value in each expression, hence we must map them to different Z3 variables. We ensure that after processing each statement the `var_version_map` is updated for any variables that may have changed by incrementing each assigned variable's version number. This ensures that future conversions from AST to Z3 formula will use different Z3 variables as necessary. Then, to determine if the formula is a tautology, we use Z3 to evaluate if the negation is unsatisfiable. If $\neg f$ is unsatisfiable, then some condition would have caused a return, and all statements afterwards are unreachable.

Algorithm 2 Remove Unreachable Code

```
Assign f to an empty Z3 formula
Collect all variable names used in the block
Initialize var_version_map as a map from variable names to integers, all set to 0
for each statement s in block's statements do
    if s is not an if-statement then
        Increment versions of each variable assigned to in s in the var_version_map
        continue
    end if
    if s contains an else branch and all blocks have unconditional returns then
        return all statements up to and including s
    end if
    l := empty list of Z3 formulas
    for each condition c in s's conditions do
        Convert c into a Z3 formula c using the var_version_map
        if c's associated block contains an unconditional return then
            f := f ∨ (¬l_0 ∧ ¬l_1 ∧ ... ∧ ¬l_n ∧ c), where n is the length of l
        end if
        Push c onto l
    end for
    Increment versions of each variable assigned to in s in the var_version_map
    if Z3 deems ¬f unsatisfiable then
        return all statements up to and including s
    end if
end for
return unchanged block
```

Branch Elimination. User assumptions may result in branches with trivial conditions like `if (true)` or `if (false)` which can be further simplified. When encountering if-statements where the truth values of the conditions are explicitly known, ProofFrog takes the following steps for canonicalization:

- A branch where the condition is `false` has its condition and associated block removed. If there are no else-if or else branches, then the if-statement can be removed entirely.
- A branch where the condition is `true` has all subsequent else-if and else blocks removed.
- If the first condition in an if-statement is `true`, then the if-statement in its entirety can be replaced with the

contents of the if-statement's first block.
- If all prior conditions have been determined to be `false`, and only an `else` block remains, the if-statement is replaced with the contents of the `else` block.

Unnecessary Field Removal. After removing branches, it is possible that some fields used in the program become unnecessary in that they cannot effect the output of any oracle's return statement. Whenever this is the case we can remove the field and any statements that reference the field without changing the game's overall behaviour. ProofFrog achieves this by walking through the statements in a block in reverse, maintaining a list of necessary variables which is initially empty. All variables used in return statement are added to this list. In addition, whenever a variables from this is assigned to, all variables used in the computation are also added to the necessary variable list. Finally, when handling nested blocks such as if-statements, any variables mentioned in the conditions are treated as necessary and the process begins recursively on all nested blocks. ProofFrog will concatenate necessary variable lists after walking through all the methods defined in a game; any field that does not appear in this list can be deemed unnecessary. After collecting the list of unnecessary fields, ProofFrog will remove each field from the game's AST and remove any statements that the field occurs in. In the case of nested statements like if-statements, the transformer will remove at the highest level of specificity possible. For example, if the field is part of a condition, that condition and associated block are removed. Whereas, if the field only appears in one of the if-statement's blocks, then just that statement from that block will be removed.

Branch Collapsing. Removing unnecessary fields can result in if-statements where multiple conditions execute the same block of code; in such cases ProofFrog will ensure that a repeated block of code only appears once. The transformer applies the following AST manipulations:

- Two adjacent if or else-if conditions with the same block can be collapsed into one block, where the new condition is the logical disjunction of the previous two conditions.
- If an if/else-if condition is adjacent to an `else` block which executes the same block of code, then that if/else-if condition and its associated block can be removed, and collapsed into the `else` block.
- If all conditions within an if-statement execute the same block of code `b`, and the if-statement contains an `else` clause, then the entire statement can be transformed into `if (true) { b }`.

Each of these manipulations only apply so long as the boolean conditions in question do not contain function calls. If a condition contains a function call, then it may have side effects, and hence collapsing multiple blocks together could change the behaviour of the program. Assuming that the conditions do not contain function calls, then the first transformation preserves behaviour because the block will be executed if either of the two conditions are satisfied. The second transformation preserves behaviour because in both cases, the block will be executed if all prior conditions are evaluated to be `false`. Finally, the third transformation preserves behaviour because in either representation, the if-statement will always execute the block `b`.

VI. Other Functionalities

A. Induction Arguments

In the basic example given in Listing 3, ProofFrog simply checks each pair of games listed in the sequence for interchangeability or indistinguishability. More complex proofs may require hybrid arguments involving a variable number of games each of which is parameterized by a counter. ProofFrog supports this via an `induction` block. In order to verify a hybrid argument in an `induction` block, ProofFrog will proceeds as follows.

First, it will verify the base case of the induction. To do so, ProofFrog will create an AST corresponding to the first game in the block where the induction variable is substituted with its starting value. This AST, after applying the standard canonicalization procedures, must be interchangeable with the AST created from the game immediately prior to the induction.

Next, ProofFrog will check that each pair of games listed inside the `induction` block are indistinguishable or interchangeable. For these checks, the induction counter variable `i` is left untouched when instantiating the games, since the hop should verify for a general `i` value. ProofFrog will then check the inductive step: it will ensure that the last game in the block (instantiated with induction variable `i`) is interchangeable with the first game in the block instantiated with `i + 1`.

Finally, ProofFrog will check the ending case of the induction. To do so ProofFrog will create an AST corresponding to the last game in the block where the induction variable is substituted with its ending value. This AST must be interchangeable with the AST created from the game listed immediately after the induction. Often, the ending value will be defined in the `let` section (say, `q`) with an explicit assumption that the adversary makes less than `q` calls to the game's oracles. Verifying each of these hops is what allows ProofFrog to ensure the correctness of a hybrid argument. The source code for a proof using an induction argument (described informally in Section VII-B) is given in Appendix B.

B. Condition Equivalence

ProofFrog also supports a method to verify hops outside of direct AST comparison. If two ASTs differ solely with respect to the conditions in their if-statements, then ProofFrog will attempt to use Z3 to check equivalence between any if-statements with differing conditions. To do so, it will attempt to create Z3 formulas for each pair of differing conditions c_1 and c_2. If either condition uses types unsupported by Z3, then the attempt will immediately abort and the hop (and hence the overall proof) will be rejected. If Z3 formulas can be created for the conditions c_1 and c_2, then ProofFrog will use Z3 to evaluate the formula $\neg(c_1 == c_2)$. If this formula is deemed unsatisfiable, then we can conclude that c_1 and c_2 are actually equivalent, and move on to the next condition. Otherwise,

if the formula is satisfiable, or Z3 deems the satisfiability unknown, then the hop is rejected. If all conditions are deemed equivalent by Z3, then we conclude the hop is valid, even though the ASTs are not strictly equivalent. This additional logic surrounding equivalence of conditions has been useful when verifying some steps in proofs using hybrid arguments.

C. Type Checking

ProofFrog also supports type checking functionality to ensure that user-written proofs are well-formed. While the ProofFrog language does not have a formal semantics defined, the type checking ensures basic properties of correctness including but not limited to:

- Variable definition before use.
- Correct typing of operators.
- Correct typing and arity of function arguments.
- Game hops and reductions providing and using the expected oracles.
- Operations on `BitString` types such as slicing and concatenation accounting for the length parameterization.

Even when ProofFrog cannot verify a user's proof via its automatic canonicalizations, it can still be of use in checking that a proof is syntactically and semantically well-formed.

VII. Case Studies

This section describes some of the more challenging proofs which ProofFrog was able to verify in *The Joy of Cryptography* [14] and the techniques the engine uses during verification. A more complete list containing formalized primitives, security definitions, and verified proofs is given in Appendix A.

A. CCA security of Encrypt-then-MAC

The encrypt-then-MAC (EtM) construction builds a CCA-secure symmetric encryption scheme from a CPA-secure symmetric encryption scheme and a secure message authentication code (MAC). EtM's encryption algorithm consists of encrypting the message with the CPA-secure scheme and then tagging this ciphertext with the MAC, and the decryption algorithm simply verifies the tag before decrypting the ciphertext. The indistinguishability-style security definition for MACs consists of a real game with oracles `GetTag` and `CheckTag` which allow one to tag a message and check whether a tag is valid for a given message respectively, and a fake game where the `CheckTag` oracle returns `true` if and only if the message had previously been provided to `GetTag`; an adversary could only distinguish between these two games by forging a MAC.

At a high level, the proof argues that since the MAC is unforgeable, EtM's decryption algorithm would be neglibly different by just returning `false` for all queries. From there, a reduction to the CPA-security of the underlying scheme completes the proof. Along with the standard transformations such as statement sorting, copy propagation, and tuple expansion, this proof applies the less frequent transformations which detect duplicated fields and unreachable code. After applying a reduction to the MAC's fake security game when generating and checking tags, the proof engine is able to detect that the set used for maintaining which ciphertexts have been returned from the encryption oracle and the set used for maintaining which messages have been tagged have identical values. Then, the decryption oracle is guarded by two checks: it returns `null` if the ciphertext was returned from the encryption oracle or if the tag was not generated through the `getTag` oracle. Since these sets are the same, the decryption oracle always returns `null`, and all code after these guards is unreachable. Removing this unreachable code then allows the proof engine to verify the reduction to CPA-security.

B. One-Time Secrecy implies CPA security

This proof demonstrates that if a public-key encryption scheme has one-time-secrecy then it is CPA-secure. ProofFrog leverages its support for induction arguments (Section VI-A) to verify this proof. At a high level, the proof consists of hopping through q games, where in game i the first i queries return the left ciphertext, and the remaining queries return the right ciphertext. Game i hops to game $i+1$ by reducing to the one-time-secrecy indistinguishability property in the $(i+1)$st oracle response. Since the adversary makes polynomially many calls, q is polynomially bounded and hence this series of reductions still gives the adversary negligible distinguishing advantage. In addition to the use of the inductive argument and the standard transformations, this proof necessitates the use of the branch elimination, unnecessary fields, and branch collapsing transformations. These transformations allow us to verify the "ramp-on" and "ramp-off" conditions of the induction. The initial left and right CPA games contain no branching whatsoever, whereas all the intermediate games require branching and internal counters to determine whether the left or right message should be encrypted. To verify the hop from the left CPA game to the first intermediate game, we supply ProofFrog with the assumption that the internal counter of queries will never be negative, from which it can eliminate the branches and fields handling any encryptions of the right ciphertext; similar techniques apply when verifying the hop from the last intermediate game to the right CPA game. The source code for this proof is given in Appendix B.

VIII. Discussion and Future Work

We are aware that ProofFrog is yet another proof verification tool in an already crowded ecosystem. In addition, while ProofFrog approaches verification with different techniques and a different philosophy from other tools, there may be an understandable apprehension for practitioners to adopt an unproven engine. One interesting avenue for future work would be an export functionality that encodes ProofFrog's transformations into more established engines. This functionality could serve to unify some of the disparate tooling within the community, provide stronger confidence in ProofFrog itself, and allow ProofFrog to function as a simpler front-end interface to more complex tools.

Another potential future avenue for work would be to expand the variety of examples that ProofFrog has been tested on. There is a substantial gap between the complexity

of proofs in textbooks versus those presented in research papers. It would be worthwhile to investigate which classes of cryptographic papers have proofs or proof steps that could be handled by the transformations developed for ProofFrog.

Even without attempting to verify a game-hopping proof, there may also be value in a user-friendly tool for specifying cryptographic definitions, games, and reductions in an accessible domain-specific language and being able to apply type-checking like in Section VI-C.

ACKNOWLEDGEMENTS

We would like to thank Wendy Lu for providing feedback on ProofFrog's usability and for formalizing many of the examples given in Appendix A. Thanks also goes to Karolin Varner for some helpful discussions throughout ProofFrog's development. D.S. was supported by Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery grant RGPIN-2022-03187.

REFERENCES

[1] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao, and B. Parno, "SoK: Computer-aided cryptography," in *2021 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 24–27, 2021, pp. 777–795.

[2] B. Blanchet, "Modeling and verifying security protocols with the applied pi calculus and ProVerif," *Foundations and Trends in Privacy and Security*, vol. 1, pp. 1–135, 10 2016.

[3] S. Meier, B. Schmidt, C. Cremers, and D. Basin, "The Tamarin prover for the symbolic analysis of security protocols," in *Computer Aided Verification*, N. Sharygina and H. Veith, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 696–701.

[4] V. Shoup, "Sequences of games: a tool for taming complexity in security proofs," Cryptology ePrint Archive, Report 2004/332, 2004. [Online]. Available: https://eprint.iacr.org/2004/332

[5] M. Bellare and P. Rogaway, "The security of triple encryption and a framework for code-based game-playing proofs," in *Advances in Cryptology – EUROCRYPT 2006*, ser. Lecture Notes in Computer Science, S. Vaudenay, Ed., vol. 4004. St. Petersburg, Russia: Springer, Berlin, Heidelberg, Germany, May 28 – Jun. 1, 2006, pp. 409–426.

[6] G. Barthe, B. Grégoire, S. Heraud, and S. Zanella Béguelin, "Computer-aided security proofs for the working cryptographer," in *Advances in Cryptology – CRYPTO 2011*, ser. Lecture Notes in Computer Science, P. Rogaway, Ed., vol. 6841. Santa Barbara, CA, USA: Springer, Berlin, Heidelberg, Germany, Aug. 14–18, 2011, pp. 71–90.

[7] B. Blanchet, "A computationally sound mechanized prover for security protocols," in *2006 IEEE Symposium on Security and Privacy*. Berkeley, CA, USA: IEEE Computer Society Press, May 21–24, 2006, pp. 140–154.

[8] R. Canetti, A. Stoughton, and M. Varia, "EasyUC: Using EasyCrypt to mechanize proofs of universally composable security," in *CSF 2019: IEEE 32nd Computer Security Foundations Symposium*, S. Delaune and L. Jia, Eds. Hoboken, NJ, USA: IEEE Computer Society Press, Jun. 25–28, 2019, pp. 167–183.

[9] C. Abate, P. G. Haselwarter, E. Rivas, A. Van Muylder, T. Winterhalter, C. Hritcu, K. Maillard, and B. Spitters, "SSProve: A foundational framework for modular cryptographic proofs in coq," in *CSF 2021: IEEE 34th Computer Security Foundations Symposium*, R. Küsters and D. Naumann, Eds. Virtual Conference: IEEE Computer Society Press, Jun. 21–24, 2021, pp. 1–15.

[10] D. Baelde, S. Delaune, C. Jacomme, A. Koutsos, and J. Lallemand, "The squirrel prover and its logic," *ACM SIGLOG News*, vol. 11, no. 2, pp. 62–83, 2024.

[11] D. A. Basin, A. Lochbihler, and S. R. Sefidgar, "CryptHOL: Game-based proofs in higher-order logic," Cryptology ePrint Archive, Report 2017/753, 2017. [Online]. Available: https://eprint.iacr.org/2017/753

[12] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Protzenko, A. Rastogi, N. Swamy, S. Zanella-Béguelin, K. Bhargavan, J. Pan, and J. K. Zinzindohoue, "Implementing and proving the TLS 1.3 record layer," in *2017 IEEE Symposium on Security and Privacy*. San Jose, CA, USA: IEEE Computer Society Press, May 22–26, 2017, pp. 463–482.

[13] M. Fischlin and A. Mittelbach, "An overview of the hybrid argument," Cryptology ePrint Archive, Report 2021/088, 2021. [Online]. Available: https://eprint.iacr.org/2021/088

[14] M. Rosulek, *The Joy of Cryptography*, Jan. 2021. [Online]. Available: https://joyofcryptography.com

[15] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, T. Rathnayake, S. Vig, B. E. Granger, R. P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M. J. Curry, A. R. Terrel, v. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, and A. Scopatz, "SymPy: symbolic computing in Python," *PeerJ Computer Science*, vol. 3, p. e103, Jan. 2017.

[16] L. M. de Moura and N. S. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340.

[17] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[18] A. B. Kahn, "Topological sorting of large networks," *Commun. ACM*, vol. 5, pp. 558–562, 1962.

APPENDIX

A. List of Primitive and Proof Case Studies in ProofFrog

The following primitives, security definitions, and proofs have been included in the ProofFrog example files.

Primitives and Associated Security Definitions.

- Symmetric Encryption Schemes [14, Definition 2.1]
  - Correctness [14, Definition 2.2]
  - One-Time Uniform Ciphertexts [14, Definition 2.5]
  - One-Time Secrecy [14, Definition 2.6]
  - CPA-security [14, Definition 7.1]
  - CPA$-security [14, Definition 7.2]
  - CCA-security [14, Definition 9.1]
  - CPA$-security [14, Definition 9.2]
- Pseudorandom Generators (PRGs) and security [14, Definition 5.1]
- Pseudorandom Functions (PRFs) and security [14, Definition 6.1]
- Message Authentication Codes (MACs) [14, Definition 10.1] and security [14, Definition 10.2]
- Public Key Encryption Schemes [14, Chapter 15]
  - Correctness [14, Chapter 15]
  - One-Time Secrecy [14, Definition 15.4]
  - CPA-security [14, Definition 15.1]
  - CPA$-security [14, Definition 15.2]

Completed Proofs.

- A symmetric encryption scheme that encrypts twice with a one-time-pad using independent keys has one-time uniform ciphertexts. [14, Claim 2.13].
- If a symmetric encryption scheme has one-time uniform ciphertexts, then it has one-time secrecy. [14, Theorem 2.15]

- If a symmetric encryption scheme Σ has one-time secrecy, then a symmetric encryption scheme which encrypts by returning a pair of ciphertexts (c_1, c_2) where $c_i = \Sigma.\text{Enc}(k_i, m)$ also has one-time secrecy. [14, Exercise 2.13]
- A symmetric encryption scheme Σ has one-time secrecy if and only if an encryption of a provided message with a one-time key is indistinguishable from an encryption of a random message with a one-time key. [14, Exercise 2.14]
- A symmetric encryption scheme Σ has one-time secrecy if and only if the ciphertext pair (c_L, c_R) is indistinguishable from the ciphertext (c_R, c_L) where m_L and m_R are encrypted with one-time keys. [14, Exercise 2.15]
- The Pseudo-OTP symmetric encryption scheme which uses a secure pseudo-random generator G to encrypt messages as $G(k) \oplus m$ provides one-time secrecy. [14, Claim 5.4]
- A length-tripling PRG which, when given a seed s, uses a length-doubling PRG G to compute $x \parallel y = G(s)$, $u \parallel v = G(y)$ and returns $x \parallel u \parallel v$ is secure assuming G's security. [14, Claim 5.5]
- Given a length-tripling PRG G, a PRG H which, when given a seed s, computes $x \parallel y \parallel z = G(s)$ and returns $G(x) \parallel G(z)$ is secure. [14, Exercise 5.8.a]
- Given a length-tripling PRG G, a PRG H which, when given a seed s, computes $x \parallel y \parallel z = G(s)$ and returns $x \parallel y$ is secure. [14, Exercise 5.8.b]
- Given a length-tripling PRG G, a PRG H which, when given a seed s, computes $x = G(s)$, $y = G(0^\lambda)$ and returns $x \oplus y$ is secure. [14, Exercise 5.8.e]
- Given a length-tripling PRG G, a PRG H which, when given a seed $s_L \parallel s_R$, computes $x = G(s_L)$, $y = G(s_R)$ and returns $x \oplus y$ is secure. [14, Exercise 5.8.f]
- Given a length-doubling PRG G, a PRG H which, when given a seed s, computes $x \parallel y = G(s)$, $w = G(y)$ and returns $(x \oplus y) \parallel w$ is secure. [14, Exercise 5.10]
- If a symmetric encryption scheme is CPA$-secure, then it is also CPA-secure. [14, Claim 7.3]
- A symmetric encryption scheme has CPA security if and only if encryptions of provided messages using the same key are indistinguishable from encryptions of random messages using the same key. [14, Exercise 7.13]
- If a symmetric encryption scheme is CCA$-secure, then it is also CCA-secure. [14, Exercise 9.6]
- If Σ is a CPA-secure symmetric encryption scheme and M is a secure MAC, then the encrypt-then-MAC construction is CCA-secure. [14, Claim 10.10]
- If a public-key encryption scheme has one-time secrecy, then it is also CPA-secure. [14, Claim 15.5]
- If Σ_{sym} is a one-time-secret symmetric-key encryption scheme and Σ_{pub} is a CPA-secure, then hybrid encryption which generates a one-time symmetric key, encrypts the symmetric key under Σ_{pub}, encrypts the message under the one-time symmetric key, and returns the pair of ciphertexts is a CPA-secure public-key encryption scheme. [14, Claim 15.9]
- If Σ_S and Σ_T are symmetric encryption schemes, where Σ_T has one-time uniform ciphertexts, then the encryption scheme Σ which encrypts a message first with Σ_S, and then encrypts the resulting ciphertext with Σ_T, also has one-time uniform ciphertexts.
- If Σ_S and Σ_T are symmetric encryption schemes, where Σ_T is CPA$-secure, then the encryption scheme Σ which encrypts a message first with Σ_S, and then encrypts the resulting ciphertext with Σ_T, is also CPA$-secure.

B. Induction Argument Source Code

The one-time secrecy definition for public key encryption schemes is given in Listing 4. The source code for the proof that a one-time secret public key encryption scheme is also CPA$-secure is given in Listing 5. The primitive source code for the public key encryption scheme and the CPA$ security definition are omitted for brevity.

C. Notes on Syntax

The syntax of ProofFrog is highly similar to C or Java. The grammar files `Primitive.g4`, `Scheme.g4`, `Game.g4`, and `Proof.g4` can be found in the repository https://github.com/ProofFrog/ProofFrog/tree/main/proof_frog/antlr. Additional notes helpful for understanding the examples provided in this paper include:

- The `<-` operation denotes uniform random sampling.
- The syntax `challenger.` is used for challenger oracle calls.
- A type `T?` represents an optional type which can contain the value `null` or a value of type `T`.

```
import 'examples/Primitives/PubKeyEnc.primitive';

Game Left(PubKeyEnc E) {
  E.PublicKey pk;
  E.SecretKey sk;
  Int count;

  E.PublicKey Initialize() {
    E.PublicKey * E.SecretKey k = E.KeyGen();
    pk = k[0];
    sk = k[1];
    count = 0;
    return pk;
  }

  E.Ciphertext? Challenge(E.Message mL, E.Message mR) {
    E.Ciphertext? result = None;
    count = count + 1;
    if (count == 1) {
      result = E.Enc(pk, mL);
    }
    return result;
  }
}

Game Right(PubKeyEnc E) {
  E.PublicKey pk;
  E.SecretKey sk;
  Int count;

  E.PublicKey Initialize() {
    E.PublicKey * E.SecretKey k = E.KeyGen();
    pk = k[0];
    sk = k[1];
    count = 0;
    return pk;
  }

  E.Ciphertext? Challenge(E.Message mL, E.Message mR) {
    E.Ciphertext? result = None;
    count = count + 1;
    if (count == 1) {
      result = E.Enc(pk, mR);
    }
    return result;
  }
}

export as OneTimeSecrecy;
```

Listing 4: One-time secrecy security definition for public key encryption schemes.

```
import 'examples/Primitives/PubKeyEnc.primitive';
import 'examples/Games/PubKeyEnc/CPA.game';
import 'examples/Games/PubKeyEnc/OneTimeSecrecy.game';

Reduction R(PubKeyEnc E, Int h) compose OneTimeSecrecy(E) against CPA(E).Adversary {
  Int count;
  E.PublicKey pk;
  E.PublicKey Initialize(E.PublicKey one_time_pk) {
    pk = one_time_pk;
    count = 0;
    return pk;
  }
  E.Ciphertext Challenge(E.Message mL, E.Message mR) {
    count = count + 1;
    if (count < h) {
      return E.Enc(pk, mR);
    } else if (count == h) {
      return challenger.Challenge(mL, mR);
    } else {
      return E.Enc(pk, mL);
    }
  }
}

proof:

let:
  Set MessageSpace;
  Set CiphertextSpace;
  Set PubKeySpace;
  Set SecretKeySpace;
  Int q;
  PubKeyEnc E = PubKeyEnc(MessageSpace, CiphertextSpace, PubKeySpace, SecretKeySpace);

assume:
  OneTimeSecrecy(E);
  calls <= q;

theorem:
  CPA(E);

games:
  CPA(E).Left against CPA(E).Adversary;
  assume R(E, 1).count >= 1;
  assume OneTimeSecrecy(E).Left.count == 1;
  induction(i from 1 to q) {
    OneTimeSecrecy(E).Left compose R(E, i) against CPA(E).Adversary;
    OneTimeSecrecy(E).Right compose R(E, i) against CPA(E).Adversary;
    assume OneTimeSecrecy(E).Left.count == 1;
    assume OneTimeSecrecy(E).Right.count == 1;
  }
  assume R(E, q).count < q + 1;
  assume OneTimeSecrecy(E).Right.count == 1;
  CPA(E).Right against CPA(E).Adversary;
```

Listing 5: Proof that a one-time-secret public key encryption scheme is also CPA-secure.