ProofFrog: A Tool For Verifying Game-Hopping Proofs

by

Ross Evans

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2024

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

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.

This thesis presents ProofFrog: a new tool for verifying cryptographic game-hopping proofs. ProofFrog is designed with the average cryptographer in mind, using an imperative syntax 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 detail the domain-specific language developed for use with the ProofFrog proof engine as well as present a sequence of worked examples that demonstrate ProofFrog's capacity for verifying proofs and the exact transformations it applies to canonicalize ASTs. 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.

Acknowledgements

There are many people who have helped me throughout my academic career so far. First and foremost, I would like to thank my supervisor Dr. Douglas Stebila. Douglas's kindness, support, guidance, and intellect have all helped me immensely in my development as a researcher. I would also particularly like to thank Douglas for being accommodating as I pursued teaching opportunities throughout my degree. I could not have asked for a better supervisor. In addition, I would like to thank my readers Dr. Mohammad Hajiabadi and Dr. Ondřej Lhoták for their time and feedback. Furthermore, I would like to thank Dr. Matthew McKague for some helpful discussions that shaped this tool's development in its early stages, and Karolin Varner for interesting discussions on Python, cryptographic proofs, and presentation skills.

I would also like to thank my friends; their significance to me is more than I could put into words. To the roomies: Kiran, Kevin, Mesha, and Carlo, for so much laughter and support. To Josh and Mary, for reminding me that there's life outside academia. And to my grad school friends: Kris, Zahra, and Ed for the many trips to the CnD, and the few trips to Lazeez, throughout this journey. Thanks in addition to Dr. Brad Lushman for the many conversations about careers, teaching, and Nintendo.

I would like to thank my family for their longstanding support as my plans have changed a multitude of times over the years.

Finally, I would like to thank Gillian for all of her love and support throughout my degree. You have been here every step of the way and I could not imagine having done this without you. Your presence brightens the darkest of my days and for that I am eternally grateful. I love you.

Dedication

This thesis is dedicated to Dooley: the best dog a boy could ever ask for.

Table of Contents

List of Figures

Chapter 1

Introduction

Secure cryptography has become an integral part of daily life. The average internet user implicitly trusts that their online activities are conducted in a confidential manner. This confidentiality, however, can be compromised in a multitude of ways spanning from an inaccurate implementation of a cryptographic construction to flaws in the underlying mathematics of the construction itself. The provable security methodology attempts to make theoretical guarantees regarding how difficult it would be for an adversary to compromise a particular construction. These guarantees usually require assuming that some particular problem is computationally expensive to solve and then demonstrating that compromising the construction is only negligibly easier than just performing the expensive computation. This methodology does not consider real world implementations of constructions; its sole purpose is to ensure that in an abstract model of computation, the adversary has no efficient strategy to compromise the construction.

Unfortunately, security proofs can be both tedious and error-prone. There are many examples of proofs that were reviewed and published only for flaws in the proof to be found much later [13]. An erroneous proof can have dire consequences if the construction is widely used. Additionally, erroneous proofs can erode confidence in the cryptographic community as a whole. It is therefore in the best interests of researchers to ensure that their proofs are correct; one promising area of research to ensure correctness of security proofs is via the use of proof assistants. These tools allow a user to specify a proof and have the steps formally checked by a computer. A certificate of correctness from such a proof assistant can give cryptographers greater confidence in the security of their constructions than a manual review from a human would, so long as the proof assistant itself is trusted to be correct.

1.1 Computer-Aided Cryptography

There are a variety of tools used in the cryptographic community for formal verification with each targeting different objectives. A systematization of knowledge paper sorts existing tools into three groups: those that focus on verifying implementation-level security, those that focus on verifying functional correctness and efficiency of implementations, and those that focus on verifying design-level security [3]. The first group of tools, which focus on implementation-level security, are primarily concerned with protecting against side-channel attacks, which attempt to compromise security by targeting computational side effects such as timing behaviour and memory access patterns. In an insecure implementation, an adversary may be able to observe computational side effects and then correlate them with some underlying secret data and hence compromise the construction. The second group of tools, which focus on functional correctness and efficiency, attempt to ensure that implementations of cryptographic constructions match their design specifications. This work is necessary as cryptographic code is rarely a direct translation of the design specification; because of cryptographic code's prevalence in modern computing, it is often heavily optimized and written in low-level, memory-unsafe languages. The third group of tools, which focus on verifying design-level security, concentrate on checking the validity of the security proofs previously mentioned.

The design-level security category is further broken down in the paper into tools that work in the symbolic model, and tools that work in the computational model. In the symbolic model, cryptographic protocols operate on abstract, atomic data called terms. The definitions of cryptographic primitives relate terms to define an equational theory, where the adversary is required to derive new terms solely from these equational theories. ProVerif [7] and Tamarin [15] are both tools operating in the symbolic model that have been used to analyze real-world protocols. In contrast, the computational model treats data as bitstrings and treats adversaries as probabilistic Turing machines. Most tools in the computational model aim to verify game-hopping proofs, which is a formalism that defines 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.

There are a variety of tools that exist for the verification of game-hopping proofs. Two of the most popular are EasyCrypt [4] and CryptoVerif [6]. EasyCrypt utilizes an imperative language for specifying games and a formula language for specifying probabilistic relational Hoare logic judgments which define security properties. To actually write a proof in EasyCrypt requires use of their tactic language which allows one to manipulate game syntax and apply logical rules to manipulate the formula being proved. These proofs are developed interactively within the Proof General interface, which can show the user the effect each tactic has when applied to the current formula, as well as display any subgoals remaining to be proved (which may happen if a proof by cases is used) [2]. As such, the proof itself functions somewhat like a final transcript of the cryptographer's session interacting with Proof General, where any mistakes or missteps have been erased. 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, like 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 Proof General. CryptoVerif leans in the opposite direction: rather than focusing on expressiveness, it focuses on automatability. It allows users to specify games via a process calculus syntax that is akin to that of functional programming languages. Proofs can often be discovered automatically via the underlying 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.

1.2 Contributions

This thesis introduces ProofFrog[1]: a new tool for verifying game-hopping cryptographic 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. An AST is a data structure commonly used when developing interpreters and compilers. Parsing a program's source code yields a parse tree which is a direct representation of the program according to its grammar. A compiler developer will then typically strip unnecessary information collected during parsing to yield an AST (for example, semicolons in many C-like languages are only necessary for parsing and do not affect the behaviour

[1] The implementation of ProofFrog is available at https://github.com/ProofFrog/ProofFrog

of the program). ASTs often also store additional context for programs that may be generated as a compiler checks for correctness or performs optimizations. The idea of using ASTs to reason about cryptographic game-hopping proofs and some of the transformations used in manipulating ASTs were first prototyped in pygamehop, which was a tool for specifying game-hopping proofs in a subset of python that ceased development in 2022 [14]. 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 each AST into a canonical form. If each pair of ASTs in a game hop can be made equivalent, then our proof engine can assert the validity of the hop. 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. Furthermore, it performs transformations to the ASTs with little user guidance, which makes writing a proof in many cases as simple as just specifying the hops. Finally, the proof syntax attempts to mimic closely to that of a typical pen-and-paper proof.

Although ProofFrog utilizes game-hopping proofs, it does not solely reside inside of the computational model. While ProofFrog does support the use of bitstrings to represent data in cryptographic primitives, it also has the ability to manipulate abstract data without a specified representation, like in the symbolic model. Furthermore, it can represent cryptographic primitives both as probabilistic algorithms like in the computational model, and as black-boxes like in the symbolic model. An author can utilize either of these two representations simply by changing the details of their proof. An author can choose to write their proof by specifying games that operate on bitstrings and defining algorithms for cryptographic primitives, or they can choose to write their proof by specifying games that operate on abstract types along with abstract primitives. ProofFrog occupies an interesting space, in that it borrows elements from both the symbolic model and the computational model, while focusing on ease of use and automatability.

ProofFrog's strategy—determining that two games are output equivalent—is an undecidable problem in the general case. As such, ProofFrog instead attempts to solve a subset of manageable cases that will expand as the tool is further developed. It is also limited in its scope as compared to other tools: it has no ability to express or prove arbitrary mathematical statements like is possible in EasyCrypt. Rather, ProofFrog attempts to automate some of the repetitive arguments found in cryptographic proofs that are both mechanically verifiable yet subtle enough for a cryptographer to potentially make errors. There will undoubtedly be many proofs that ProofFrog will be unable to verify in its current form and that is to be expected. The hope is that the simplicity of using ProofFrog when compared to existing tools will encourage a wider adoption of proof assistant technology in a field where correctness is paramount. Future development of ProofFrog could utilize it as a platform to implement a variety of helpful features for cryptographers, such as the ability to perform type-checking of proofs even without verification, or the ability to export proofs automatically to LaTeX diagrams that could be included in publications.

The remainder of the thesis proceeds as follows. Chapter 2 presents relevant background and notation for cryptographic proofs. Chapter 3 presents details of ProofFrog's domain-specific language as well as some of the criteria that went into its development. The bulk of the thesis is written in Chapter 4. It presents a sequence of worked examples that highlight a variety of different proof techniques that ProofFrog supports. The examples are presented in the same order in which they were implemented for the proof engine, with increasing complexity. Accompanying each example is a diagram showing the functionality of the engine developed up to that point. Finally, Chapter 5 discusses potential avenues for future work on ProofFrog.

Chapter 2

Cryptographic Proofs

Modern cryptographic proofs attempt to establish the security of a construction under precise assumptions and conditions. In doing so, an author will provide a security definition, which details the goal of an adversary as well as their computational capabilities. To demonstrate that a particular construction satisfies a particular security definition, the author must prove that all efficient adversaries can only succeed with a negligibly higher probability than desired.

There are many techniques that one can utilize to prove security definitions, but this thesis will focus specifically on security definitions expressed as pairs of indistinguishable games and proofs using the game-hopping proof technique. We present the details of our formalism first, and then connect this to the broader literature at the end of the chapter. In our approach, 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 $\Pr[G \circ A \to b']$ to denote the probability that the adversary outputs the bit b' when given access to G's oracles. We can then precisely define what it means for a pair of games to be indistinguishable.

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*** *(with notation $G_L \equiv G_R$) 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 $G_L \approxeq G_R$) 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.*

The game-hopping proof technique then allows an author to prove indistinguishability of two games G_L and G_R in a security definition by first providing a sequence of games G_L, G_0, G_1, G_2, ..., G_R, and then proving each adjacent pair of games in the sequence to be either interchangeable or indistinguishable. By the fact that both interchangeability and indistinguishability are transitive, and the fact that interchangeability implies indistinguishability, proving each adjacent pair of games to be indistinguishable suffices to prove that G_L and G_R are indistinguishable.

Proving interchangeability of games requires showing that the oracles behave the same under all possible inputs. Such arguments can often be made using logical and syntactic features of the code. For example, if two games have identical code apart from one defining a variable that is never used, it is easy to argue that the two games are interchangeable. On the other hand, proving indistinguishability for a pair of games can be more challenging. One useful strategy is to apply a reduction. Reductions act simultaneously as both games and adversaries, and they allow an author to leverage the indistinguishability of simpler primitives in more complicated proofs. As an example, assume that we wish to demonstrate that $G_L \approxeq G_R$, with the assumption that $H_L \approxeq H_R$. To utilize a reduction, the author would write a game R such that $G_L \equiv H_L \circ R$. That is, R must utilize the oracles provided by H_L to exactly mimic the behaviour of G_L. Then, we note that by associativity of composition, for an adversary $\mathcal{A}$, we have that $(H_L \circ R) \circ \mathcal{A}$ is the same program as $H_L \circ (R \circ \mathcal{A})$. That is, we can consider $(H_L \circ R)$ to be a game providing oracles to the adversary $\mathcal{A}$, or we can consider H_L to be a game providing oracles to the adversary $(R \circ \mathcal{A})$. By the indistinguishability of H_L, we have that $\mathrm{Adv}(R \circ \mathcal{A}, H_L, H_R)$ is negligible, allowing us to hop to $H_R \circ (R \circ A)$. Applying associativity again yields $(H_R \circ R) \circ \mathcal{A}$. This sequence of manipulations let us show that $G_L \equiv (H_L \circ R) \approxeq (H_R \circ R)$. If $(H_R \circ R) \equiv G_R$ then the proof is done, otherwise the author would write further games applying reductions or interchangeability arguments to reach G_R.

Game-hopping as a proof technique and the precise details of the accompanying formalism have been presented in a variety of ways. Shoup, in a noteworthy tutorial paper, presented game-hopping as a method to simplify cryptographic proofs, where the games themselves are probability spaces operating on random variables [19]. Unlike our description of games as packages of code, this paper does not enforce any explicit syntax for the description of the games, rather just using a combination of mathematical notation and exposition to explain the argument. Shoup describes three types of hops between games:

1. Transitions based on indistinguishability, which we have previously described.
2. Bridging steps, which rewrite the game while preserving the output distributions. This is equivalent to our concept of interchangeability.
3. Transitions based on failure events, which are also sometimes referred to as "identical-until-bad" arguments. This type of argument demonstrates indistinguishability by first showing that the output distributions of two games are the same unless some `bad` variable gets set to `true`, and then showing that the probability `bad` gets set to `true` is negligible. ProofFrog does not natively support this type of argument, and so we will not use it for any of our proofs.

In another popular approach, Bellare and Rogaway explicitly treated games as programs with oracles written as procedures [5]. They presented arguments using both pseudocode and a formalized programming language for games, with the suggestion that such a formalized language could prove helpful in the automated verification of game-hopping proofs. Recently, Brzuska et al. have introduced the notion of the state-separable proof [9]. This formalism suggests taking games and splitting them into individual packages which contain collections of oracles and associated state. The main purpose behind decomposing games in this manner is that with clearly defined state boundaries, operations on games such as reductions can be expressed using solely algebraic manipulations. It can also make generic some arguments that would otherwise need to be repeated for each proof. This formalism is also used extensively in Mike Rosulek's textbook *The Joy of Cryptography*, from which this thesis will verify a number of examples [18]. The arguments made in this thesis are ultimately based in the state-separable proof formalism, however we will refrain from delving too deeply into the algebraic notation that is used in the original paper.

The formalism used by Brzuska et al., Rosulek, and this thesis requires all security definitions to be written in terms of pairs of indistinguishable games. However, not all works in the broader literature follow this approach. For example, in Katz and Lindell's *Introduction to Modern Cryptography*, security definitions are written by providing the definition of a single game where the adversary is trying to achieve some win condition [12]. The construction is secure if and only if the adversary can only win with a negligibly higher probability than desired. Despite the apparent differences between these approaches, one can convert a security definition based on a win condition into a security definition based on indistinguishability. For example, some security definitions may be described as "hidden-bit" games, in which the challenger uniformly randomly generates a bit, the adversary queries oracles that in some way depend on this bit, and then the adversary wins if they can correctly determine which bit the challenger generated. There is a simple transformation from this type of security definition into one written in terms of indistinguishability: simply write a pair of games where one always sets the bit to 0 and the other always sets the bit to 1. If an adversary can distinguish between these two games with a probability that is non-negligibly greater than $\frac{1}{2}$, then the same strategy can also be used to determine the bit in the "hidden-bit" formulation, and vice versa. This transformation demonstrates that the two security definitions are actually equivalent.

Hidden-bit games are just one example of a possible win condition that happens to be particularly amenable for transforming into an indistinguishability format. Another common type of win condition requires the adversary to solve a search problem. For example, constructions that provide authentication should not be forgeable, therefore a security definition for such a construction will define the adversary $\mathcal{A}$ to have won the game G if they can produce a valid forgery. We can also write an equivalent security definition utilizing indistinguishability like so: define two games G_{real} and G_{fake} that behave like G but also each provide an oracle `CheckForgery`, which takes as input $\mathcal{A}$'s attempted forgery. The G_{real} game will actually perform the check, returning `true` if the forgery was successful and `false` otherwise. The G_{fake} game will always just return `false` to indicate that the forgery failed. The adversary can distinguish between these two games with a non-negligible probability if and only if they can perform a valid forgery, which results in an indistinguishability security definition that is equivalent to the win condition security definition. In general, this transformation is applicable for any type of win condition security definition for a game G and a win condition w. We can simply write two games G_{real} and G_{fake} identical to G apart from a `CheckWin` oracle, where G_{real} checks if w is satisfied and G_{fake} always returns `false`. Just as before, the adversary can only distinguish between G_{real} and G_{fake} by achieving the win-condition. This generic transformation demonstrates that security definitions written in terms of pairs of indistinguishable games are no less powerful than security definitions written in terms of win conditions.

Chapter 3

Domain-Specific Language for Cryptographic Proofs

The first step in the development of ProofFrog was to create a domain-specific language in which users can represent games, cryptographic constructions, and proofs. While implementing the proof engine, we considered leveraging the syntax and parser of an already existing language like Python to enhance usability. But, we found that most proofs require additional context beyond the syntax provided by existing languages. In order to supply a proof with enough context while utilizing an existing language would require the user to learn how to write additional annotations to their source code specifically for ProofFrog. Furthermore, the ASTs of existing languages are often large with support for a variety of different programming paradigms and many types of syntactic sugar. The main strategy employed by the proof engine is AST comparison, and as such, the fewer different ways there are to write the same piece of code, the better. This wide diversity of syntax made it difficult to narrow down the acceptable syntax for proofs, and it could potentially restrict authors from programming in a particular style if it was unsupported by the proof engine. Given these constraints, we thought a domain-specific language would be the easiest for both the proof engine and for the users. This approach ensures that ProofFrog has all the information necessary for verifying a proof just by building the AST. Furthermore, users are not required to learn additional unintuitive syntax that is tacked onto a preexisting language.

There were a few design criteria we aimed for when writing the grammar for ProofFrog's domain-specific language. First, we initially decided that the grammar itself should be based on the C family of languages. It can easily be assumed that the average cryptographer will have at one point learned a language in the C family, which makes the syntax for writing proofs easy to pick up and understand. Secondly, we tried to avoid a large divergence between the syntax used for specifying games and constructions versus the syntax used for expressing the proofs, so as to reduce the mental load when switching between writing definitions and writing proofs. Finally, we aimed for the proof syntax to mimic how a pen-and-paper proof would be written: namely, the user should be able to define some variables, list some assumptions, state the security property they wish to prove, and then provide a sequence of games that proves the property. The grammar itself was written in ANTLR 4, which allowed for rapid prototyping and development [17].

ProofFrog supports four different types of files: primitive files, scheme files, game files, and proof files. Proof files are the only type of file that ProofFrog actually verifies; the remaining exist to provide definitions of security definitions and schemes to be used in proofs. Examples for the syntax of each file will be presented as necessary in Chapter 4; for now, we simply describe the purpose of each file and a few key features of their syntax. The full grammar for each of the file types, in ANTLR 4 syntax, is provided in Appendix A.

3.1 Primitive Files and Scheme Files

Primitive files and scheme files both serve a related purpose: they describe the sets and functions that are associated with a particular mathematical object. The key difference is that primitives model abstract constructions whereas schemes are concrete. Primitive files allow a user to provide just method signatures for each function the object defines, whereas scheme files require defining each function with both a signature and an implementation. Each scheme models a particular type of primitive: for example, one might define a public key encryption scheme primitive and an accompanying RSA scheme. In this way, primitives and schemes map closely to the concepts of abstract and concrete classes in traditional object-oriented programming.

3.2 Game Files

Game files allow the user to specify security definitions in the form of a pair of games that must be proved indistinguishable. Each function listed in a game is provided implicitly to the adversary in the form of an oracle. Games may also contain private state that is maintained between oracle calls. Finally, each game has the option to specify an `Initialize` method which is assumed to be implicitly called before the adversary program starts. `Initialize` also has the ability to return information to the adversary that they may use during the attack (for example, adversaries should be provided the public key when attacking public key encryption).

3.3 Proof Files

Proof files allow the user to specify any reductions or intermediate games that may be used in the proof, after which the user must define statements in four sections: the `let` section, the `assume` section, the `theorem` section, and a sequence of games. The `let` section details the variables, primitives, and schemes to be used in the assumptions, theorem, and proof. The `assume` section allows the user to specify indistinguishability assumptions for schemes defined in the `let` section. The `theorem` section states which security definition is going to be proved for which scheme. Finally, the `games` section lists a sequence of games, starting with one of the games from the pair specified in the theorem and ending with the other game. Each game in the sequence is checked to be either interchangeable with the next game, or to be indistinguishable with the next game. Indistinguishability only occurs if the author writes a reduction to utilize a previously specified indistinguishability assumption.

Chapter 4

Building Up ProofFrog

This chapter will detail the capabilities of ProofFrog as it was initially developed. To do so, we will present a sequence of worked examples that ProofFrog can verify, starting with basic examples and moving to those that necessitate more complex strategies. In each section we will detail the steps of the proof, how the definitions and proof steps map into our domain specific language, and finally the transformations that ProofFrog must apply to each game's AST in order to verify the result. The proofs and definitions (aside from a couple in Section 4.2) are adapted from Rosulek's textbook *The Joy of Cryptography*, with minimal changes [18]. The text served as a useful collection of proofs with which to validate ProofFrog against, since each security definition and proof it models is written in the state-separable proof framework using pairs of indistinguishable games.

4.1 CPA$ Security Implies CPA Security

The first proof this chapter will start with focuses on the properties of symmetric encryption schemes, which is a type of construction used for securing communication of messages between two parties with a shared secret. Symmetric encryption schemes are a fundamental tool and are often the first definition introduced when learning cryptography. The proof detailed in this chapter, that CPA$ security implies CPA security, is one that compares related security definitions for a symmetric encryption scheme so as to better understand their relative strength. Beyond the fact that understanding the strength of security definitions is inherently useful, this proof is also a good choice for first discussion due to its simplicity. The proof essentially is a straightforward application of the general game-hopping technique that was previously described, i.e, using a sequence of games that are interchangeable or indistinguishable, and using reductions to leverage prior indistinguishability assumptions. As such, it was a natural proof with which to begin the development of the ProofFrog engine and will allow us to detail the basic steps taken in ProofFrog's game canonicalization strategy.

4.1.1 Definitions

Definition 5 (Definition 2.1 from [18])**.** *A symmetric-key encryption scheme consists of a key space* K*, a message space* M*, a ciphertext space* C*, and the following algorithms:*

- KeyGen*: a randomized algorithm that outputs a key* $k \in \mathrm{K}$*.*
- Enc*: a (possibly randomized) algorithm that takes a key* $k \in \mathrm{K}$ *and a plaintext* $m \in \mathrm{M}$ *as input, and outputs a ciphertext* $c \in \mathrm{C}$*.*
- Dec*: a deterministic algorithm that takes a key* $k \in \mathrm{K}$ *and ciphertext* $c \in \mathrm{C}$ *as input, and outputs a plaintext* $m \in \mathrm{M}$*, or the symbol* None *to indicate failure to decrypt.*

When referring to the entire scheme by a single variable Σ*, the components are denoted by* Σ.KeyGen*,* Σ.Enc*,* Σ.Dec*,* Σ.K*,* Σ.M*, and* Σ.C

The next definition will provide a pair of games for a security definition. Both games maintain a secret key as part of their state and provide an oracle named CTXT to which the adversary can pass a message m. The real game encrypts the provided message with the key and returns the ciphertext, whereas the random game responds with a randomly sampled ciphertext. These games will be modelled via pseudocode. The game's name appears as the header in bold. Statements after the header occur as part of the game's initialization. If a line is underlined, that indicates it is an oracle: the name of the oracle is provided, with arguments indicated in parentheses, and all code belonging to the oracle is indented. The pseudocode notation should be largely intuitive: for now, simply note that we use := to indicate assignment and ← to indicate sampling uniformly randomly from a set.

Definition 6 (Definition 7.2 from [18])**.** *A symmetric encryption scheme* Σ *is CPA\$ secure if and only if* $\textbf{\textit{CPA\$}}^{\Sigma}_{real} \approx \textbf{\textit{CPA\$}}^{\Sigma}_{rand}$ *where:*

```
$\mathbf{CPA\$}^{\Sigma}_{\text{real}}$

$k := \Sigma.\text{KeyGen}()$

$\text{CTXT}(m \in \Sigma.\text{M})$:
    $c := \Sigma.\text{Enc}(k, m)$
    return $c$
```

```
$\mathbf{CPA\$}^{\Sigma}_{\text{rand}}$

$\text{CTXT}(m \in \Sigma.\text{M})$:
    $c \leftarrow \Sigma.\text{C}$
    return $c$
```

Intuitively, CPA$ security captures the notion that encrypted messages "look random", in that, without knowledge of the secret key of the challenger, the adversary should not be able to distinguish ciphertexts generated by encrypting messages from ciphertexts that are chosen uniformly randomly. For CPA$ security to hold, this property must be true even when the adversary can encrypt as many messages as they would like containing whatever content they choose. The initialism CPA comes from "chosen plaintext attack", since the adversary is supplying the messages to the challenger. We will now present a definition for CPA security, in which both games also maintain a secret key but provide a different oracle called Eavesdrop to the adversary. In each game the Eavesdrop oracle accepts two messages; the games differ in which message is encrypted with the secret key. The name "Eavesdrop" comes from the imagined scenario in which the adversary is eavesdropping on a channel where ciphertexts are being transmitted.

Definition 7 (Definition 7.1 from [18])**.** *A symmetric encryption scheme Σ is CPA secure if and only if $\boldsymbol{CPA}^{\Sigma}_{L} \approx \boldsymbol{CPA}^{\Sigma}_{R}$ where:*

```
$\mathbf{CPA}^{\Sigma}_{L}$

$k := \Sigma.\text{KeyGen}()$

$\text{Eavesdrop}(m_L, m_R \in \Sigma.\text{M})$:
    $c := \Sigma.\text{Enc}(k, m_L)$
    return $c$
```

```
$\mathbf{CPA}^{\Sigma}_{R}$

$k := \Sigma.\text{KeyGen}()$

$\text{Eavesdrop}(m_L, m_R \in \Sigma.\text{M})$:
    $c := \Sigma.\text{Enc}(k, m_R)$
    return $c$
```

CPA security models that ciphertexts do not leak information about plaintexts. If a scheme is CPA secure, then the distinguishing advantage between the two games is negligible, and so an adversary cannot determine which plaintext the outputted ciphertext corresponds to after calling the Eavesdrop oracle. As before, the adversary can repeat this experiment as many times as they would like with whichever messages they choose. Intuitively, if all ciphertexts generated by a symmetric encryption scheme “look random”, then one would expect that ciphertexts also do not leak any information about their corresponding plaintexts. This intuition can be formalized in the following theorem.

4.1.2 Theorem and Proof

Theorem 1 (Claim 7.3 from [18])**.** *If a symmetric encryption scheme Σ is CPA\$ secure then it is also CPA secure.*

Proof. To prove that Σ is CPA secure, we must show a sequence of games from $\mathbf{CPA}_L^\Sigma$ to $\mathbf{CPA}_R^\Sigma$ where each pair of adjacent games in the game hop sequence is either interchangeable or indistinguishable. To begin, we will present a short overview of the game hops to expect:

1. $\mathbf{CPA}_L^\Sigma$

- We write a reduction $\mathbf{R}_1^\Sigma$ that uses the CTXT oracle provided by the real CPA\$ game to perfectly mimic the behaviour of the left CPA game. This is an interchangeable hop.

2. $\mathbf{CPA\$}_{\text{real}}^\Sigma \circ \mathbf{R}_1^\Sigma$

- The premise of the proof assumes that either CPA\$ game can be substituted for the other. This is an indistinguishable hop.

3. $\mathbf{CPA\$}_{\text{rand}}^\Sigma \circ \mathbf{R}_1^\Sigma$

- We write a new reduction $\mathbf{R}_2^\Sigma$ that passes m_R as an argument to the CTXT oracle instead of m_L, which will not matter when composed with the random game. This is an interchangeable hop.

4. $\mathbf{CPA\$}_{\text{rand}}^\Sigma \circ \mathbf{R}_2^\Sigma$

- Once again, we can substitute either CPA\$ game for the other. This is an interchangeable hop.

5. $\mathbf{CPA\$}_{\text{real}}^\Sigma \circ \mathbf{R}_2^\Sigma$

- The previous game exactly mimics the behaviour of the right CPA game. This is an interchangeable hop.

6. $\mathbf{CPA}_R^\Sigma$

Now, we show each game hop in full detail. The first hop will utilize a reduction: $\mathbf{CPA}_L^\Sigma \equiv \mathbf{CPA\$}_{\text{real}}^\Sigma \circ \mathbf{R}_1^\Sigma$

```
$\mathbf{CPA\$}_{\text{real}}^\Sigma$

$k := \Sigma.\text{KeyGen}()$

$\text{CTXT}(m \in \Sigma.\text{M})$:
    $c := \Sigma.\text{Enc}(k, m)$
    return $c$
```

```
$\mathbf{R}_1^\Sigma$

$\text{Eavesdrop}(m_L, m_R \in \Sigma.\text{M})$:
    $c :=$ challenger.CTXT$(m_L)$
    return $c$
```

The reduction simply uses the CTXT oracle provided by $\mathbf{CPA\$}_{\text{real}}^\Sigma$. The two games $\mathbf{CPA}_L^\Sigma$ and $\mathbf{CPA\$}_{\text{real}}^\Sigma \circ \mathbf{R}_1^\Sigma$ are interchangeable because the Eavesdrop oracle behaves identically for both games: it just returns the encryption of m_L. The second hop utilizes the assumption that Σ is CPA\$ secure and so we can replace the real version of the CPA game that encrypts messages with the random version that returns random ciphertexts. Hence, we have that $\mathbf{CPA\$}_{\text{real}}^\Sigma \circ \mathbf{R}_1^\Sigma \approx \mathbf{CPA\$}_{\text{rand}}^\Sigma \circ \mathbf{R}_1^\Sigma$.

```
$\mathbf{CPA\$}_{\text{rand}}^\Sigma$

$\text{CTXT}(m \in \Sigma.\text{M})$:
    $c \leftarrow \Sigma.\text{C}$
    return $c$
```

```
$\mathbf{R}_1^\Sigma$

$\text{Eavesdrop}(m_L, m_R \in \Sigma.\text{M})$:
    $c :=$ challenger.CTXT$(m_L)$
    return $c$
```

The third hop notes that the argument m is unused in the CTXT oracle. We then can write a new reduction $\mathbf{R}_2^\Sigma$ that passes m_R to CTXT instead of m_L. Note that this hop is actually one which is interchangeable despite using reductions. The interchangeability is because the behaviour of a program remains the same if you change an unused argument. This hop demonstrates that $\mathbf{CPA\$}_{\text{rand}}^\Sigma \circ \mathbf{R}_1^\Sigma \equiv \mathbf{CPA\$}_{\text{rand}}^\Sigma \circ \mathbf{R}_2^\Sigma$

```
$\mathbf{CPA\$}^{\Sigma}_{\text{rand}}$

CTXT($m \in \Sigma$.M):
    $c \leftarrow \Sigma$.C
    return $c$
```

```
$\mathbf{R}^{\Sigma}_{2}$

Eavesdrop($m_L, m_R \in \Sigma$.M):
    $c :=$ challenger.CTXT($m_R$)
    return $c$
```

The fourth hop once again applies the assumption that Σ is CPA\$ secure. This assumption allows us to revert back to using real ciphertexts instead of random ciphertexts in the challenger's CTXT oracle. This hop demonstrates that $\mathbf{CPA\$}^{\Sigma}_{\text{rand}} \circ \mathbf{R}^{\Sigma}_{2} \approx \mathbf{CPA\$}^{\Sigma}_{\text{real}} \circ \mathbf{R}^{\Sigma}_{2}$.

```
$\mathbf{CPA\$}^{\Sigma}_{\text{real}}$

$k :=$ Σ.KeyGen()

CTXT($m \in \Sigma$.M):
    $c := \Sigma$.Enc($k, m$)
    return $c$
```

```
$\mathbf{R}^{\Sigma}_{2}$

Eavesdrop($m_L, m_R \in \Sigma$.M):
    $c :=$ challenger.CTXT($m_R$)
    return $c$
```

And finally, since the behaviour of this reduction is just to return the encryption of m_R, $\mathbf{CPA\$}^{\Sigma}_{\text{rand}} \circ \mathbf{R}^{\Sigma}_{2} \equiv \mathbf{CPA}^{\Sigma}_{R}$. Hence, if Σ is CPA\$ secure, then

$$\mathbf{CPA}^{\Sigma}_{L} \equiv \mathbf{CPA\$}^{\Sigma}_{\text{real}} \circ \mathbf{R}^{\Sigma}_{1} \approx \mathbf{CPA\$}^{\Sigma}_{\text{rand}} \circ \mathbf{R}^{\Sigma}_{1} \equiv \mathbf{CPA\$}^{\Sigma}_{\text{rand}} \circ \mathbf{R}^{\Sigma}_{2} \approx \mathbf{CPA\$}^{\Sigma}_{\text{real}} \circ \mathbf{R}^{\Sigma}_{2} \equiv \mathbf{CPA}^{\Sigma}_{R}$$

By transitivity, $\mathbf{CPA}^{\Sigma}_{L} \approx \mathbf{CPA}^{\Sigma}_{R}$ and so Σ is CPA secure. □

4.1.3 ProofFrog Encoding

To write this proof in ProofFrog requires first implementing all relevant definitions in the domain-specific language and then writing a proof file listing out the sequence of games. The symmetric encryption scheme definition can be found in Figure 4.1. In the definition, we provide the method signatures expected for symmetric encryption schemes, and store the sets with which the primitive is constructed. Storing the sets allows them to be accessible later on and used for typing of variables in games. The game files to model both CPA and CPA\$ security are found in Figure 4.2 and Figure 4.3 respectively. With all of these defined, our proof file consists first of writing the reductions $\mathbf{R}^{\Sigma}_{1}$ and $\mathbf{R}^{\Sigma}_{2}$, which is detailed in Figure 4.4. Syntactically, reductions have the same bodies as games. In the declaration of a reduction, however, one must also specify what challenger the reduction acts as an adversary for (listed after compose) and what type of adversary is querying the reduction (list after against). This extra information would allow a type checker to ensure that the reduction is calling valid challenger oracles and that the oracles defined by the reduction have the correct signatures for the adversary to call. Having written the reductions, the proof file is concluded by completing the four sections of a proof as written in Figure 4.5.

```
Primitive SymEnc(Set MessageSpace, Set CiphertextSpace, Set KeySpace) {
    Set Message = MessageSpace;
    Set Ciphertext = CiphertextSpace;
    Set Key = KeySpace;

    Key KeyGen();
    Ciphertext Enc(Key k, Message m);
    Message Dec(Key k, Ciphertext c);
}
```

Figure 4.1: ProofFrog syntax for a symmetric encryption scheme primitive.

```
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;
```

Figure 4.2: ProofFrog syntax for the pair of games modelling CPA security of a symmetric encryption scheme.

```
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$;
```

Figure 4.3: ProofFrog syntax for the pair of games modelling CPA$ security of a symmetric encryption scheme.

```
Reduction R1(SymEnc E) compose CPA$(E) against CPA(E).Adversary {
    E.Ciphertext Eavesdrop(E.Message mL, E.Message mR) {
        return challenger.CTXT(mL);
    }
}

Reduction R2(SymEnc E) compose CPA$(E) against CPA(E).Adversary {
    E.Ciphertext Eavesdrop(E.Message mL, E.Message mR) {
        return challenger.CTXT(mR);
    }
}
```

Figure 4.4: Reductions used to prove Theorem 1.

```
proof:
let:
    Set M;
    Set C;
    Set K;
    SymEnc SE = SymEnc(M, C, K);
assume:
    CPA$(SE);
theorem:
    CPA(SE);
games:
    CPA(SE).Left against CPA(SE).Adversary; // Game 1
    // Game 1 -> Game 2, by interchangeability.
    CPA$(SE).Real compose R1(SE) against CPA(SE).Adversary; // Game 2
    // Game 2 -> Game 3, by indistinguishability
    CPA$(SE).Random compose R1(SE) against CPA(SE).Adversary; // Game 3
    // Game 3 -> Game 4, by interchangeability
    CPA$(SE).Random compose R2(SE) against CPA(SE).Adversary; // Game 4
    // Game 4 -> Game 5, by indistinguishability
    CPA$(SE).Real compose R2(SE) against CPA(SE).Adversary; // Game 5
    // Game 5 -> Game 6, by interchangeability
    CPA(SE).Right against CPA(SE).Adversary; // Game 6
```

Figure 4.5: Proof file to prove Theorem 1.

For this proof, we use arbitrary sets `M`, `C`, and `K` to denote the `MessageSpace`, `CiphertextSpace` and `KeySpace` of `SE` since the proof works independently of the sets that the symmetric encryption scheme uses. These games model computation on abstract data from arbitrary sets and as such are an example of a proof influenced by the symoblic model; we are using abstract data for our messages rather than explicitly defining messages as bitstrings. After indicating the assumption that `SE` satisfies CPA$ security and that the proof is for CPA security of `SE`, the list of games is specified in the same order as in the given pen-and-paper proof. One purposeful aspect of ProofFrog's design is that this list matches one-to-one with the the pen-and-paper proof given: the intent is that matching ProofFrog syntax so closely to the pen-and-paper proof will make it easy for the average cryptographer to use.

4.1.4 Validating Indistinguishability

Verifying this proof requires two main functionalities: checking that indistinguishability assumptions are used correctly, and performing composition of games with reductions to check interchangeability. To detect if an indistinguishability assumption is used in a particular hop, we simply check whether the following four criteria are true:

1. Both steps in the hop use a reduction.
2. The steps are identical apart from changing which game the reduction is composed with.
3. The game the reduction is composed with is changed to its corresponding pair in the security definition.
4. The security definition pair appears with identical parameters in the `assumptions` section of the proof.

If all of these criteria are true, then this step is valid by indistinguishability. For example, the second hop, from `CPA$(SE).Real compose R1(SE)` to `CPA$(SE).Random compose R1(SE)` 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$(SE).Random compose R1(SE)` to `CPA$(SE).Random compose R2(SE)` must be checked as interchangeable because the reduction being applied is not the same in each step.

4.1.5 Verifying Interchangeability

Validating the interchangeability steps is more challenging as the engine must transform steps involving reductions into single games. Recall that the strategy ProofFrog uses to check interchangeability is to simplify game ASTs into a canonical form and then check equality at the AST level. Validating a hop like `CPA(SE).Left` to `CPA$(SE).Real compose R1(SE)` requires taking the definitions of `CPA$(SE).Real` and `R1(SE)` and composing them into a single game.

The first step is undertaken by the InstantiationTransformer. Its purpose is to 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 SE = SymEnc(M, C, K). ProofFrog will associate with the value SE a copy of the SymEnc primitive AST, where any references to Message or MessageSpace are replaced with M, any references to Ciphertext or CiphertextSpace are replaced with C, etc. The InstantiationTransformer is also applied for game and reduction definitions: the value CPA(SE).Left is associated with the CPA left game AST where types like E.Key are replaced with K. Function calls like E.KeyGen and E.Enc are rewritten as SE.KeyGen and SE.Enc. 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. Rewriting all parameters in terms of variables defined in the let section ensures that this mistake is avoided.

4.1.6 Creating the Inlined Game

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

The first step is to combine states together: each field included in the AST associated with CPA$(SE).Real and the AST associated with R1(SE) should be included in the inline 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 allowed during parsing of variable names, so these names are guaranteed to be conflict-free.

The second step is to combine Initialize methods together. Because the reduction also acts as an adversary, it has the possibility of receiving state from the challenger in the form of a parameter to its Initialize method. The inlined game, being just a game and not an adversary, should have no parameters in its `Initialize` method. To combine 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. This yields the following AST for `CPA$(SE).Real compose R1(SE)`:

```
Game Inlined() {
  K challenger@k;
  Void Initialize() {
    challenger@k = SE.KeyGen();
  }
  C Eavesdrop(M mL, M mR) {
    return SE.Enc(challenger@k, mL);
  }
}
```

4.1.7 Standardizing Variables Names

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$(SE).Real compose R1(SE)`, written as a single game. However, the inlined game AST and the AST for `CPA(SE).Left` still do not compare as identical, because variable names are mismatched. The inlined game uses the variable name `challenger@k` whereas `CPA(SE).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(SE).Left` and the inlined game AST to match identically, validating this proof step.

In summary, the transformations described above are sufficient to validate each interchangeable game-hop in the proof. Note that in the hop from game three to game four, where the argument m_L is replaced with m_R: this hop is easily handled by the `InlineTransformer`. Substitution of the variable name is a no-op because there are no occurrences of the argument m in the random CTXT oracle body, and inlining then substitutes the same random sample statement for both methods. The other steps using indistinguishability arguments are handled by simply checking that the assumption is properly asserted, as previously described. There are, of course, many more advanced proofs that are too complex for these simple strategies to validate. However, this proof, that CPA$ security implies CPA security, is provable under these transformations, and provides an interesting use case in the form of security definition strength comparison.

In Figure 4.6, we present a flowchart diagram of the high-level steps taken by ProofFrog's engine as it stands after the building blocks added in this initial worked example. As this chapter progresses through further worked examples, we will update this diagram with the new functionality required to validate each proof, culminating with a final diagram illustrating the functionality of the engine in full.

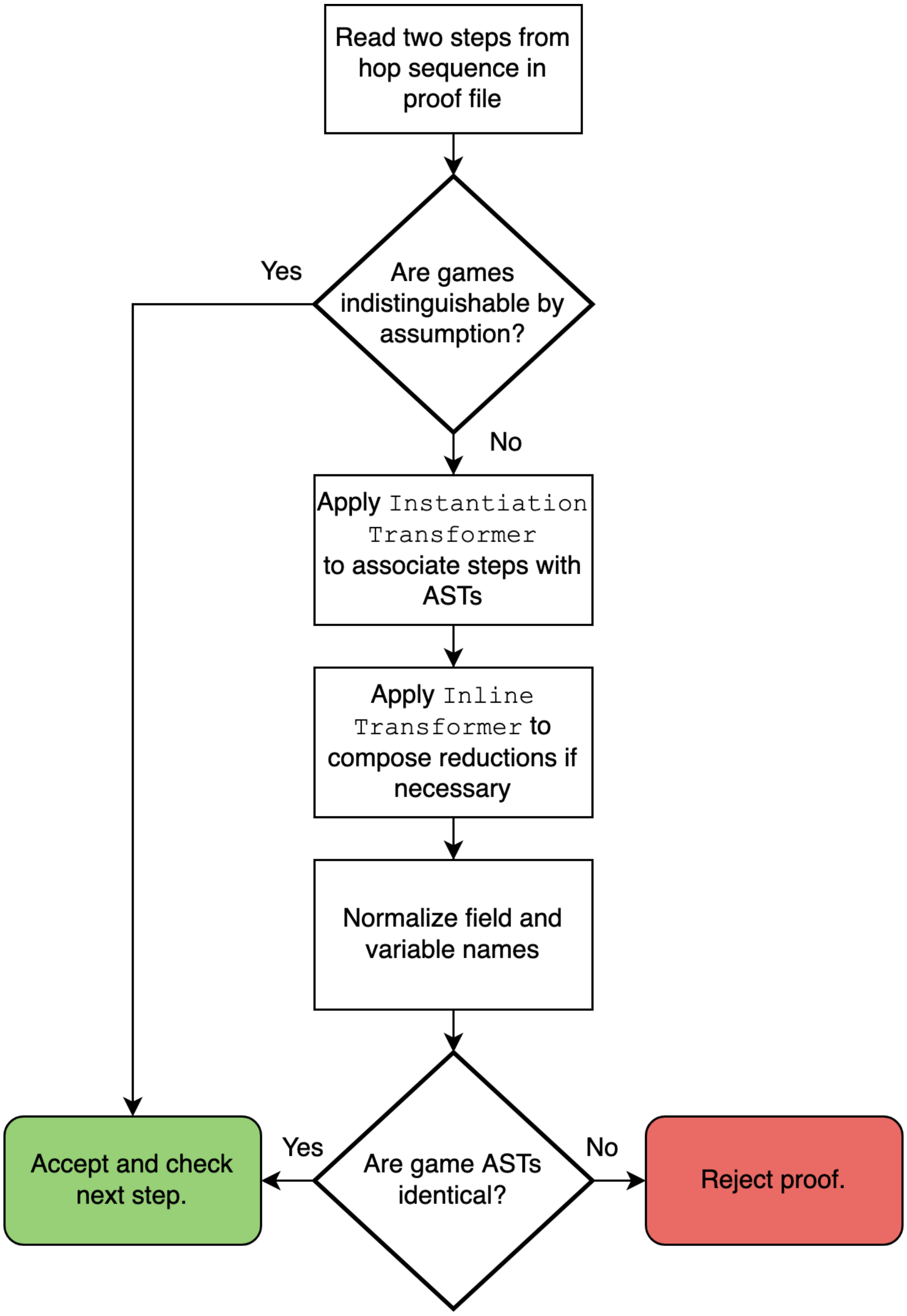
Read two steps from hop sequence in proof file
Are games indistinguishable by assumption?
Yes
No
Apply Instantiation Transformer to associate steps with ASTs
Apply Inline Transformer to compose reductions if necessary
Normalize field and variable names
Are game ASTs identical?
Yes
No
Accept and check next step.
Reject proof.

Figure 4.6: A flowchart of ProofFrog engine functionality necessary to prove Theorem 1.

4.2 Double Symmetric Encryption and One-Time Uniform Ciphertexts

The previous proof demonstrated how interchangeability can in some cases be established simply from method inlining. However, two oracles can be interchangeable while also having different ASTs after inlining. For example, one AST may have statements that are in a different ordering than the other, or one may have statements that are redundant and do not appear in the other. These differences would not affect the behaviour of the oracles and yet would result in different ASTs. This section will detail a proof where statement re-ordering, dead-code elimination, and some other source code transformations are necessary to yield two identical ASTs. The proof is of a different nature than the previous example; whereas the last proof focused on comparing two security definitions, this proof will focus on analyzing the security properties of a new symmetric encryption scheme created from two other symmetric encryption schemes. In addition to being another type of proof the average cryptographer may wish to reason about, this example will allow us to present a ProofFrog scheme file and some of the engine's more advanced game canonicalization transformations.

4.2.1 Definitions

Definition 8. *A double symmetric encryption scheme is a symmetric encryption scheme composed from two other symmetric encryption schemes, S and T, where $S.\mathrm{C} = T.\mathrm{M}$, and:*

$$\mathrm{KeyGen}() = (S.\mathrm{KeyGen}(), T.\mathrm{KeyGen}())$$

$$\mathrm{Enc}((k_S, k_T), m) = T.\mathrm{Enc}(k_T, S.\mathrm{Enc}(k_S, m))$$

$$\mathrm{Dec}((k_S, k_T), c) = S.\mathrm{Dec}(k_S, T.\mathrm{Dec}(k_T, c))$$

For this proof we will also introduce a new security definition, the one-time uniform ciphertexts property.

Definition 9 (Definition 2.5 from [18])**.** *A symmetric encryption scheme Σ has one-time uniform ciphertexts if and only if $\boldsymbol{OTUC}^{\Sigma}_{\mathit{real}} \approx \boldsymbol{OTUC}^{\Sigma}_{\mathit{rand}}$, where:*

```
$\mathbf{OTUC}^{\Sigma}_{\text{real}}$

CTXT($m \in \Sigma$.M):
    $k := \Sigma$.KeyGen()
    $c := \Sigma$.Enc($k, m$)
    return $c$
```

```
$\mathbf{OTUC}^{\Sigma}_{\text{rand}}$

CTXT($m \in \Sigma$.M):
    $c \leftarrow \Sigma$.C
    return $c$
```

The one-time uniform ciphertexts property is much like that of CPA\$ security, the difference being that the secret key is scoped locally to the CTXT oracle, and is therefore regenerated for each adversary call. The regeneration prevents the adversary from performing attacks that rely on observing relationships between multiple ciphertexts encrypted with the same key. Because the types of attacks an adversary can perform are more limited, the one-time uniform ciphertexts property is weaker than that of CPA\$ security. One can also ask, under what conditions would a double symmetric encryption scheme satisfy the one-time uniform ciphertexts property? The answer, interestingly, depends only on the security property of T, and not that of S.

4.2.2 Theorem and Proof

Theorem 2. *Assume Σ is a double symmetric encryption scheme composed from two symmetric encryption schemes S and T. If T has one-time uniform ciphertexts, then so does Σ.*

A similar result can be proved for CPA\$ security, where a double symmetric encryption scheme is CPA\$ secure (and hence CPA secure) if T is CPA\$ secure. However, the proof of that result only requires dead-code elimination and not statement reordering. Hence, the one-time uniform ciphertexts property, despite being weaker, is a better example to demonstrate ProofFrog's capabilities.

Proof. To prove that Σ has one-time uniform ciphertexts, we must show that $\mathbf{OTUC}^{\Sigma}_{\text{real}} \approx \mathbf{OTUC}^{\Sigma}_{\text{rand}}$. We again provide an overview of the sequence of games before elaborating in full.

1. $\mathbf{OTUC}^{\Sigma}_{\text{real}}$

- We write a reduction $\mathbf{R}_1^{\Sigma}$ that uses the oracles provided by the real one-time uniform ciphertexts game for T to perfectly mimic the behaviour of the real one-time uniform ciphertexts game for Σ. This is an interchangeable hop.

2. $\mathbf{OTUC}_{\text{real}}^{T} \circ \mathbf{R}_1^{\Sigma}$

- The premise of the proof assumes that the real one-time uniform ciphertexts game for T can be replaced with the random game. This is an indistinguishable hop.

3. $\mathbf{OTUC}_{\text{rand}}^{T} \circ \mathbf{R}_1^{\Sigma}$

- We inline the call to the challenger's CTXT oracle to form an intermediate game used for explanation. This is an interchangeable hop.

4. $\mathbf{G}_1^{\Sigma}$

- We argue that apart from some redundant statements, the random game and the intermediate game are identical. This is an interchangeable hop.

5. $\mathbf{OTUC}_{\text{rand}}^{\Sigma}$

We now detail each hop in full. The starting game is $\mathbf{OTUC}_{\text{real}}^{\Sigma}$. It is shown below with the definitions of Σ's key generation and encryption algorithms already inlined.

```
$\mathbf{OTUC}_{\text{real}}^{\Sigma}$
CTXT($m \in \Sigma.\text{M}$):
    $k_S := S.\text{KeyGen}()$
    $k_T := T.\text{KeyGen}()$
    $c_1 := S.\text{Enc}(k_S, m)$
    $c_2 := T.\text{Enc}(k_T, c_1)$
    return $c_2$
```

Next, we note that the T.KeyGen() statement and the $T.\text{Enc}(k_T,\ c_1)$ statement, when paired together, form the code of the real one-time uniform ciphertexts game for T. This allows us to utilize a reduction and show that $\mathbf{OTUC}_{\text{real}}^{\Sigma} \equiv \mathbf{OTUC}_{\text{real}}^{T} \circ \mathbf{R}_1^{\Sigma}$

```
$\mathbf{OTUC}^{T}_{\text{real}}$
CTXT($m \in T$.M):
    $k := T$.KeyGen()
    $c := T$.Enc($k, m$)
    return $c$
```

```
$\mathbf{R}^{\Sigma}_{1}$
CTXT($m \in \Sigma$.M):
    $k_S := S$.KeyGen()
    $c_1 := S$.Enc($k_S, m$)
    $c_2 :=$ challenger.CTXT($c_1$)
    return $c_2$
```

Note that inlining the challenger.CTXT call, renaming variables, and re-ordering statements yields $\mathbf{OTUC}^{\Sigma}_{\text{real}}$, so these two games are indeed interchangeable. Then, via the assumption that T has one-time uniform ciphertexts, we can replace the real one-time uniform ciphertexts game for T with the random one-time uniform ciphertexts game for T, and hop indistinguishably to $\mathbf{OTUC}^{T}_{\text{rand}} \circ \mathbf{R}^{\Sigma}_{1}$.

```
$\mathbf{OTUC}^{T}_{\text{rand}}$
CTXT($m \in T$.M):
    $c \leftarrow T$.C
    return $c$
```

```
$\mathbf{R}^{\Sigma}_{1}$
CTXT($m \in \Sigma$.M):
    $k_S := S$.KeyGen()
    $c_1 := S$.Enc($k_S, m$)
    $c_2 :=$ challenger.CTXT($c_1$)
    return $c_2$
```

We can inline the game and reduction together to get the following intermediate game named $\mathbf{G}^{\Sigma}_{1}$. Since the only transformation made is inlining, we have that $\mathbf{OTUC}^{T}_{\text{rand}} \circ \mathbf{R}^{\Sigma}_{1} \equiv \mathbf{G}^{\Sigma}_{1}$.

```
$\mathbf{G}^{\Sigma}_{1}$
CTXT($m \in \Sigma$.M):
    $k_S := S$.KeyGen()
    $c_1 := S$.Enc($k_S, m$)
    $c_2 \leftarrow T$.C
    return $c_2$
```

And then, we simply note that the first two statements assigning to k_S and c_1 are not used in the return value of CTXT. The behaviour of this oracle is just to return a value selected uniformly randomly from T's ciphertext space. And, since $T.\mathrm{C} = \Sigma.\mathrm{C}$, $\mathbf{G}_1^\Sigma$ just samples uniformly randomly from Σ's ciphertext space, which is the exact behaviour of $\mathbf{OTUC}_{\text{rand}}^\Sigma$. Hence, we have that

$$\mathbf{OTUC}_{\text{real}}^\Sigma \equiv \mathbf{OTUC}_{\text{real}}^T \circ \mathbf{R}_1^\Sigma \approx \mathbf{OTUC}_{\text{rand}}^T \circ \mathbf{R}_1^\Sigma \equiv \mathbf{G}_1^\Sigma \equiv \mathbf{OTUC}_{\text{rand}}^\Sigma$$

and so Σ has one-time uniform ciphertexts. □

4.2.3 ProofFrog Encoding

This proof's definition in ProofFrog is not too novel as compared to the previous section. The main difference is that we must also define a scheme file for the double symmetric encryption scheme (Figure 4.7), which we have not yet presented an example of. The `extends` syntax for the scheme file indicates which primitive this scheme models and therefore which sets and method signatures the scheme needs to provide. This scheme also demonstrates the `requires` clause, which allows a user to place constraints on a scheme's parameterization, in the form of a boolean expression, which would be validated by a type checker during instantiation. Other than that, we simply must provide our one-time uniform ciphertexts security definition (Figure 4.8), reduction (Figure 4.9), and proof steps (Figure 4.10) to complete the proof.

```
Scheme DoubleSymEnc(SymEnc S, SymEnc T) extends SymEnc {
    requires S.Ciphertext == T.Message;
    Set Message = S.Message;
    Set Ciphertext = T.Ciphertext;
    Set Key = S.Key * T.Key;
    Key KeyGen() {
        S.Key key1 = S.KeyGen();
        T.Key key2 = T.KeyGen();
        return [key1, key2];
    }
    Ciphertext Enc(Key k, Message m) {
        S.Ciphertext c1 = S.Enc(k[0], m);
        T.Ciphertext c2 = T.Enc(k[1], c1);
        return c2;
    }
    Message Dec(Ciphertext c) {
        S.Ciphertext c2 = T.Dec(k[1], c);
        S.Message m = S.Dec(k[0], c2);
        return m;
    }
}
```

Figure 4.7: ProofFrog syntax for a double symmetric encryption scheme.

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

Figure 4.8: ProofFrog syntax for the pair of games modelling the one-time uniform ciphertexts property of a symmetric encryption scheme.

```
Reduction R(DoubleSymEnc D, SymEnc S, SymEnc T)
compose OTUC(T) against OTUC(D).Adversary {
    D.Ciphertext CTXT(D.Message m) {
        S.Key k1 = S.KeyGen();
        S.Ciphertext c1 = S.Enc(k1, m);
        T.Ciphertext c2 = challenger.CTXT(c1);
        return c2;
    }
}
```

Figure 4.9: Reduction used in the proof of Theorem 2.

```
proof:
let:
    Set M;
    Set K1;
    Set K2;
    Set I;
    Set C;
    SymEnc S = SymEnc(M, I, K1);
    SymEnc T = SymEnc(I, C, K2);
    DoubleSymEnc D = GeneralDoubleSymEnc(S, T);
assume:
    OTUC(T);
theorem:
    OTUC(D);
games:
    OTUC(D).Real against OTUC(D).Adversary; // Game 1
    // Game 1 -> Game 2, by interchangeability
    OTUC(T).Real compose R(D, S, T) against OTUC(D).Adversary; // Game 2
    // Game 2 -> Game 3, by indistinguishability
    OTUC(T).Random compose R(D, S, T) against OTUC(D).Adversary; // Game 3
    // Game 3 -> Game 4, by interchangeability
    OTUC(D).Random against OTUC(D).Adversary; // Game 4
```

Figure 4.10: Proof file for Theorem 2.

4.2.4 Initial ASTs

We now aim to detail the transformations undertaken by ProofFrog in order to verify this proof. We shall begin with the first hop, from `OTUC(D).Real` to `OTUC(T).Real compose R(D, S, T)`. Simply applying the steps from the previous section: i.e, instantiating games and creating an inlined game for the reduction, is not sufficient to establish AST equivalence. The instantiation of `OTUC(D).Real` causes the method calls `D.KeyGen()` and `D.Enc()` to be replaced with those provided in the scheme definition. Doing so yields the following AST:

```
Game Real() {
  C CTXT(M m) {
    K1 D.KeyGen@key1 = S.KeyGen();
    K2 D.KeyGen@key2 = T.KeyGen();
    K1 * K2 k = [D.KeyGen@key1, D.KeyGen@key2];
    I D.Enc@c1 = S.Enc(k[0], m);
    C D.Enc@c2 = T.Enc(k[1], D.Enc@c1);
    C c = D.Enc@c2;
    return c;
  }
}
```

Whereas, the AST created from instantiating and inlining `OTUC(T).Real compose R(D, S, T)` is the following:

```
Game Inlined() {
  C CTXT(M m) {
    K1 k1 = S.KeyGen();
    I c1 = S.Enc(k1, m);
    K2 challenger.CTXT@k = T.KeyGen();
    C challenger.CTXT@c = T.Enc(challenger.CTXT@k, c1);
    C c2 = challenger.CTXT@c;
    return c2;
  }
}
```

These games are mathematically interchangeable, they differ only in the timing of when the keys are generated, and that the `Real` game creates an unnecessary tuple. Neither of these changes result in any difference in the return value. However, given just instantiation and inlining, there is no way to verify this proof. The nature of the reduction forces the ordering of function calls to be `S.KeyGen()`, `S.Enc()`, `T.KeyGen()`, `T.Enc()`, whereas the real game will always generate all keys first before performing any encryption. Since there is no way a user could write this proof so that ProofFrog would accept it via just instantiation and inlining, the only solution is to improve ProofFrog's game canonicalization abilities. ProofFrog has some further simplifications it can apply to each of these games to yield the same AST. First, the tuple `k` in the real game is unnecessary: we can rewrite `k[0]` and `k[1]` in terms of `D.KeyGen@key1` and `D.KeyGen@key2`. Second, it can remove duplicate variables like `c` in the real game, and `c2` in the inlined game. Third, it can perform reordering of statements so that keys are always generated before any encryption. These simplifications are enough to verify each of the three hops in the proof.

4.2.5 Tuple Expansion

The ExpandTupleTransformer is the transformer in ProofFrog that takes tuples and rewrites them in terms of individual variables. For example, a tuple like `Int * Int t = [a, b]` would be rewritten into `Int t@0 = a; 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 cannot be expanded. If the first condition is violated then one cannot statically determine which variable to read from or write to. And if the second condition is violated (for example, if the tuple is assigned to the result of a function call), then one cannot determine what values `t@0` and `t@1` should be set to.

For each tuple that is considered eligible, the ExpandTupleTransformer 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 semantics to the original, except `t` has been removed. Applying the ExpandTupleTransformer to the real game yields:

```
Game Real() {
  C CTXT(M m) {
    K1 D.KeyGen@key1 = S.KeyGen();
    K2 D.KeyGen@key2 = T.KeyGen();
    K1 k@0 = D.KeyGen@key1;
    K2 k@1 = D.KeyGen@key2;
    I D.Enc@c1 = S.Enc(k@0, m);
    C D.Enc@c2 = T.Enc(k@1, D.Enc@c1);
    C c = D.Enc@c2;
    return c;
  }
}
```

This form is amenable to further transformations: namely `k@0`, `k@1`, and `c` are all duplicates of previously defined variables. Removing such duplicates helps to further simplify the game ASTs.

4.2.6 Copy Propagation

Copy propagation is a technique used in compiler optimizations. In many cases, compiler transformations can introduce variables which are direct copies of others, like `K1 k@0 = D.KeyGen@key1`, for example. Copy propagation is the act of removing such direct copies and replacing them with the original definition where possible [1, Chapter 9.1.5]. Traditionally copy propagation helps prevent redundant computation at run-time, but for ProofFrog it will instead be repurposed for simplifying ASTs directly after ProofFrog's other transformations introduce duplicated variables. The `RedundantCopyTransformer` 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 semantics of the code, while removing an unnecessary duplicated variable. Applying the transformation to the real game yields the following AST:

```
Game Real() {
  C CTXT(M m) {
    K1 D.KeyGen@key1 = S.KeyGen();
    K2 D.KeyGen@key2 = T.KeyGen();
    I D.Enc@c1 = S.Enc(D.KeyGen@key1, m);
    C D.Enc@c2 = T.Enc(D.KeyGen@key2, D.Enc@c1);
    return D.Enc@c2;
  }
}
```

And applying the transformation to the inlined game yields:

```
Game Inlined() {
  C CTXT(M m) {
    K1 k1 = S.KeyGen();
    I c1 = S.Enc(k1, m);
    K2 challenger.CTXT@k = T.KeyGen();
    C challenger.CTXT@c = T.Enc(challenger.CTXT@k, c1);
    return challenger.CTXT@c;
  }
}
```

These ASTs are significantly closer to each other than before: it is clear that both are computing the same values just with different statement orderings and different variable names. Canonicalizing the statement orderings and renaming variables as done previously is sufficient to produce two identical ASTs and verify the first hop of the proof.

4.2.7 Statement Ordering and Dead Code Elimination

Previously, we had described the final transformation necessary to canonicalize the ASTs as “reordering statements so that keys are always generated before any encryptions“. Although such a reordering would suffice to verify this proof hop, it does not describe a general strategy to ensure two interchangeable ASTs have identical orderings 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:

- 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`.
- If `s` is a return statement or contains a return statement in a nested block and `t` assign to a field, then `s` depends on `t`.
- 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 `return`s (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 `return` statement is used to create a list of statements. 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. We then use Kahn's algorithm to topologically sort the list of statements created by the traversal according to the dependency graph [11]. This approach yields a canonical ordering of statements for any block that contains a return statement.

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. Applying the sorting procedure to both games followed by variable renaming yields two identical ASTs for hopping from `OTUC(D).Real` to `OTUC(T).Real compose R(D, S, T)`. The second hop applies indistinguishability and requires no further transformations. The final hop, from `OTUC(T).Random compose R(D, S, T)` to `OTUC(D).Random` requires dead-code elimination, which actually occurs as a side-effect of the sorting procedure. Any statements that do not have an effect on the return value of the oracle do not appear in the depth-first traversal, and hence do not appear in the list of statements provided to Kahn's algorithm, which results in their removal after sorting.

In totality, this proof concerning double symmetric encryption schemes can be verified by using both the strategies described previously such as instantiation, inlining, and variable renaming combined with the new strategies of tuple expansion, copy propagation,

Algorithm 1 Topological Sort

```
Require: The block's final statement is a return statement
visited_statements = empty stack
Generate dependency graph $G$ for the block
Perform depth first traversal according to $G$ starting with the return statement. Push
    to visited_statements for each statement visited.
kahn_queue = empty queue
sorted_statements = empty list
while visited_statements is not empty do
    Pop s from visited_statements
    if s has no dependencies in $G$ then
        Enqueue statement to kahn_queue
    end if
end while
while kahn_queue is not empty do
    Dequeue s from kahn_queue and append to sorted_statements
    for each statement r that depends on s in $G$ do
        Remove r's dependence on s in $G$
        if r has no dependencies then
            Enqueue r to kahn_queue
        end if
    end for
end while
return sorted_statements
```

and statement sorting. Figure 4.11 shows an updated flowchart diagram of the steps taken by ProofFrog. Note that ProofFrog will apply each transformation step repeatedly for all proofs that it verifies, stopping when no transformation can simplify the AST any further. Transformations must be specifically designed with this repeated simplification in mind, otherwise there is a risk that subsequent transformations may revert each other's changes, resulting in an infinite loop.

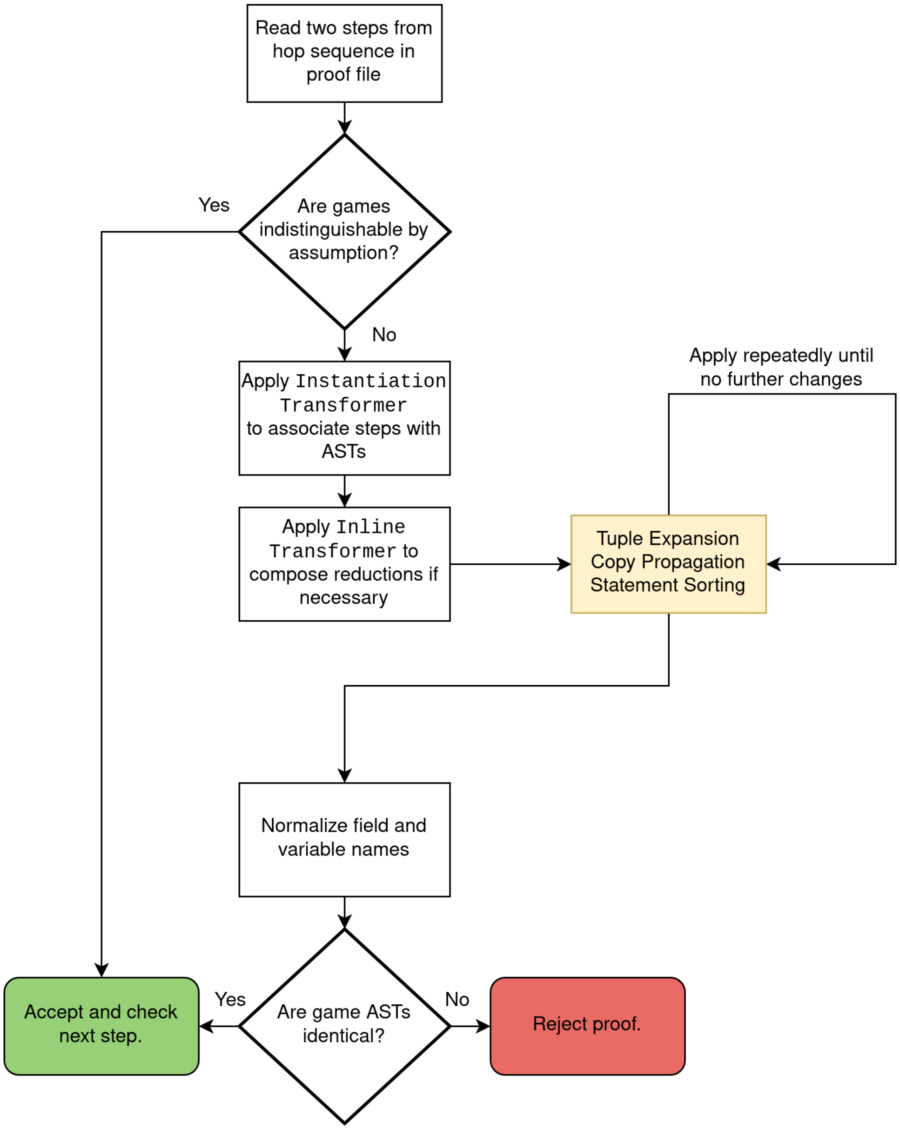
Read two steps from hop sequence in proof file
Are games indistinguishable by assumption?
Yes
No
Apply `Instantiation Transformer` to associate steps with ASTs
Apply `Inline Transformer` to compose reductions if necessary
Apply repeatedly until no further changes
Tuple Expansion
Copy Propagation
Statement Sorting
Normalize field and variable names
Are game ASTs identical?
Yes
No
Accept and check next step.
Reject proof.

Figure 4.11: A flowchart of ProofFrog engine functionality necessary to prove Theorem 2.

4.3 Constructing a Length-Tripling PRG

The previous two proofs focused on proving properties of symmetric encryption schemes. Although encryption schemes are important, they are not the sole construction useful for cryptographic proofs. The game hopping technique can also be leveraged to prove statements about simpler constructions often used in cryptography like pseudorandom generators, pseudorandom functions, and others. This section will focus on a proof concerning the construction of a new secure pseudorandom generator from an existing secure pseudorandom generator. In doing so we aim to illustrate ProofFrog's capacity to handle proofs beyond just those for encryption schemes, some new transformations to support the use of bitstrings, and some symbolic computation features built into the proof engine.

4.3.1 Definition

We provide the definition of a pseudorandom generator, which is a construction that can take a shorter bitstring as a "seed" and from that seed produce a longer bitstring, which appears to be sampled uniformly randomly from the larger space. A pseudorandom generator is secure so long as an adversary can not distinguish randomly sampled bitstrings from the pseudorandom generator's outputs.

Definition 10 (Definition 5.1 from [18])**.** *Let $G : \{0,1\}^\lambda \to \{0,1\}^{\lambda+l}$ be a deterministic function with $l > 0$. G is a* ***secure pseudorandom generator*** *(PRG) if and only if $\boldsymbol{PRG}^G_{real} \approx \boldsymbol{PRG}^G_{rand}$ where:*

```
$\mathbf{PRG}^G_{\text{real}}$

Query():
    $s \leftarrow \{0,1\}^\lambda$
    return $G(s)$
```

```
$\mathbf{PRG}^G_{\text{rand}}$

Query():
    $r \leftarrow \{0,1\}^{\lambda+l}$
    return $r$
```

The quantity l is called the "stretch" of the PRG. In the case where $l = \lambda$, we call G a length-doubling PRG. In the case where $l = 2\lambda$, we call G a length-tripling PRG, and so on. Note that G is treated as a black box function from $\{0,1\}^\lambda$ to $\{0,1\}^{\lambda+l}$. G's internal definition does not matter so long as it satisfies the indistinguishability property for secure pseudorandom generators (namely, that $\mathbf{PRG}^G_{\text{real}} \approx \mathbf{PRG}^G_{\text{rand}}$). In addition, this definition of a secure PRG differs slightly from non cryptographically secure pseudorandom number generators provided in typical programming language standard libraries, which are given an initial seed and can then generate a sequence of numbers at a user's request without any further input. In contrast, our definition of a PRG produces exactly one deterministic output for each input. The deterministic nature of the PRG is why the security game generates the seed itself rather than accepting a seed from the adversary. If the adversary $\mathcal{A}$ could provide the seed s to the security game then $\mathcal{A}$ could trivially distinguish between the real and random games by just computing $G(s)$ and comparing it to the output of $\text{Query}(s)$. Obscuring the seed from $\mathcal{A}$ is therefore necessary to ensure that the security definition is actually satisfiable.

4.3.2 Theorem and Proof

We now provide a definition of a length-tripling PRG built from a length-doubling PRG and prove its security.

Theorem 3 (Claim 5.5 from [18])**.** *Assume $G : \{0,1\}^\lambda \to \{0,1\}^{2\lambda}$ is a secure length-doubling PRG. Then H, as defined below, is a secure length-tripling PRG.*

```
$H(s \in \{0,1\}^\lambda)$:
    $x \| y := G(s)$
    $u \| v := G(y)$
    return $x \| u \| v$
```

The $\|$ operator in H indicates concatenation of bitstrings. Since $G(s)$ yields a bitstring of length 2λ, x would consist of the first λ bits of $G(s)$ and y the remaining λ bits. The same notation applies to $u\|v$. Note that the behaviour of H is defined by the provided pseudocode whereas the behaviour of G is left as a black box.

Proof. We must show that $\mathbf{PRG}^H_{\text{real}} \approx \mathbf{PRG}^H_{\text{rand}}$. We begin with a brief overview of the games:

1. $\mathbf{PRG}^H_{\text{real}}$

- We rewrite the first use of G via a reduction to G's real security game. This is an interchangeable hop.

2. $\mathbf{PRG}^G_{\text{real}} \circ \mathbf{R}^H_1$

- We use the assumed security of G to replace the real game with the random game. This is an indistinguishable hop.

3. $\mathbf{PRG}^{G}_{\text{rand}} \circ \mathbf{R}^{H}_{1}$

- We inline the game and reduction together to form an intermediate game for explanation. This is an interchangeable hop.

4. $\mathbf{G}^{H}_{1}$

- We argue that sampling a bitstring of length 2λ is equivalent to sampling two bitstrings of length λ independently. This is an interchangeable hop.

5. $\mathbf{G}^{H}_{2}$

- We rewrite the second use of G via a reduction to G's real security game. This is an interchangeable hop.

6. $\mathbf{PRG}^{G}_{\text{real}} \circ \mathbf{R}^{H}_{2}$

- We use the assumed security of G to replace the real game with the random game. This is an indistinguishable hop.

7. $\mathbf{PRG}^{G}_{\text{rand}} \circ \mathbf{R}^{H}_{2}$

- We inline the game and reduction together to form an intermediate game for explanation. This is an interchangeable hop.

8. $\mathbf{G}^{H}_{3}$

- We note that all three bitstrings we concatenate together at this point are sampled uniformly randomly, which is equivalent to sampling one bitstring of length 3λ. This is an interchangeable hop.

9. $\mathbf{PRG}^{H}_{\text{rand}}$

We now discuss each of these steps in detail. For the first game, substituting the definition of H into the real PRG game definition yields the following:

```
$\mathbf{PRG}^H_{\text{real}}$
Query():
    $s \leftarrow \{0,1\}^\lambda$
    $x \| y := G(s)$
    $u \| v := G(y)$
    return $x \| u \| v$
```

We then note that the first two lines are identical to those of $\mathbf{PRG}^G_{\text{real}}$. We can therefore use a reduction to factor out the first call to $G(s)$ in terms of G's real PRG security game. This reduction is an interchangeable hop demonstrating that $\mathbf{PRG}^H_{\text{real}} \equiv \mathbf{PRG}^G_{\text{real}} \circ \mathbf{R}^H_1$

```
$\mathbf{PRG}^G_{\text{real}}$
Query():
    $s \leftarrow \{0,1\}^\lambda$
    return $G(s)$
```

```
$\mathbf{R}^H_1$
Query():
    $x \| y :=$ challenger.Query()
    $u \| v := G(y)$
    return $x \| u \| v$
```

Then, because G is assumed to be a secure PRG, we can leverage this assumption to replace the real version of the G's security game with the random version. This is an indistinguishability hop proving that $\mathbf{PRG}^G_{\text{real}} \circ \mathbf{R}^H_1 \approx \mathbf{PRG}^G_{\text{rand}} \circ \mathbf{R}^H_1$

```
$\mathbf{PRG}^G_{\text{rand}}$
Query():
    $r \leftarrow \{0,1\}^{\lambda+\lambda}$
    return $r$
```

```
$\mathbf{R}^H_1$
Query():
    $x \| y :=$ challenger.Query()
    $u \| v := G(y)$
    return $x \| u \| v$
```

We can then perform inlining of the Query call to yield an intermediate game $\mathbf{G}^H_1$. We also simplify $\lambda + \lambda$ to 2λ. This demonstrates interchangeability of the previous reduction with this intermediate game: $\mathbf{PRG}^G_{\text{rand}} \circ \mathbf{R}^H_1 \equiv \mathbf{G}^H_1$.

```
$\mathbf{G}_1^H$
Query():
    $x\|y \leftarrow \{0,1\}^{2\lambda}$
    $u\|v := G(y)$
    return $x\|u\|v$
```

At this point, we would like to again utilize the security of G to rewrite $u\|v$ as being randomly sampled. But, in order to do so, the argument to G (which is y in this case) must be a value sampled from $\{0,1\}^\lambda$. At the moment, y represents the final λ bits after sampling from $\{0,1\}^{2\lambda}$. To support a reduction to the security of G, we can first perform another interchangeable hop to a different intermediate game $\mathbf{G}_2^H$. This new game will simply sample x and y independently: since sampling two bitstrings of length λ independently is semantically identical to sampling a bitstring of length 2λ and then subdividing it. We then have that $\mathbf{G}_1^H \equiv \mathbf{G}_2^H$. This step is fairly obvious to humans and would not typically require so much elaboration, however, the development of a proof engine requires even obvious steps to be carefully written for machine understanding.

```
$\mathbf{G}_2^H$
Query():
    $x \leftarrow \{0,1\}^\lambda$
    $y \leftarrow \{0,1\}^\lambda$
    $u\|v := G(y)$
    return $x\|u\|v$
```

Since $y \leftarrow \{0,1\}^\lambda$ now matches the statement $s \leftarrow \{0,1\}^\lambda$ modulo variable naming, we have that the definition of $u\|v$ can be rewritten in terms of a reduction to the real version of G's security game. This is an interchangeable hop: $\mathbf{G}_2^H \equiv \mathbf{PRG}_{\text{real}}^G \circ \mathbf{R}_2^H$.

```
$\mathbf{PRG}_{\text{real}}^G$
Query():
    $s \leftarrow \{0,1\}^\lambda$
    return $G(s)$
```

```
$\mathbf{R}_2^H$
Query():
    $x \leftarrow \{0,1\}^\lambda$
    $u\|v :=$ challenger.Query()
    return $x\|u\|v$
```

Just as before, we can use the fact that G is assumed to be a secure PRG, and replace the real version of G's security game with the random version. This is an indistinguishability hop proving that $\mathbf{PRG}^{G}_{\text{real}} \circ \mathbf{R}^{H}_{2} \approx \mathbf{PRG}^{G}_{\text{rand}} \circ \mathbf{R}^{H}_{2}$.

```
$\mathbf{PRG}^{G}_{\text{rand}}$
Query():
    $r \leftarrow \{0,1\}^{\lambda+\lambda}$
    return $r$
```

```
$\mathbf{R}^{H}_{2}$
Query():
    $x \leftarrow \{0,1\}^{\lambda}$
    $u \| v :=$ challenger.Query()
    return $x \| u \| v$
```

We can once again simplify $\lambda + \lambda$ to 2λ and inline to get the final intermediate game: $\mathbf{G}^{H}_{3}$. This is interchangeable with the previous game: $\mathbf{PRG}^{G}_{\text{rand}} \circ \mathbf{R}^{H}_{2} \equiv \mathbf{G}^{H}_{3}$.

```
$\mathbf{G}^{H}_{3}$
Query():
    $x \leftarrow \{0,1\}^{\lambda}$
    $u \| v \leftarrow \{0,1\}^{2\lambda}$
    return $x \| u \| v$
```

Finally, we note that sampling a bitstring of length λ and concatenating it with a sampled bitstring of length 2λ produces the same result as sampling a bitstring of length 3λ. And, because the behaviour of $\mathbf{PRG}^{H}_{\text{rand}}$ is to return a randomly sampled bitstring of length 3λ, we have that $\mathbf{G}^{H}_{3} \equiv \mathbf{PRG}^{H}_{\text{real}}$. This final hop completes the sequence and demonstrates that $\mathbf{PRG}^{H}_{\text{real}} \approx \mathbf{PRG}^{H}_{\text{rand}}$, and so H is a secure PRG.

□

4.3.3 ProofFrog Encoding

Writing this proof in ProofFrog consists of the usual steps. First, we write a primitive file to model PRGs (Figure 4.12). Second, we write a scheme file to implement the length-tripling PRG construction (Figure 4.13). The ProofFrog language does not support multiple variable declarations in a single line, so when calling `G.evaluate(s)`, we extract the first λ bits into `x` and the remaining λ bits into `y` via a slice syntax akin to Python's. Third, we write the PRG security definition as a pair of games in Figure 4.14. In the pen-and-paper proof we often alternated between two equivalent syntaxes: sometimes sampling two bitstrings of length l_1 and l_2 and concatenating them, other times sampling one bitstring of length $l_1 + l_2$. We make these transitions by using reductions to and from a pair of bitstring sampling games in Figure 4.15. The underlying mechanics of this work the same as any other reduction; it is simply a special case of using a reduction to show interchangeability instead of indistinguishability. The use of a reduction is a slight workaround; in an ideal world the proof engine would be able to recognize that bitstrings can be sampled independently without the use of these extra steps. But for now, this workaround makes verification simpler for the proof engine at the expense of the author doing some extra work. Finally, we list the reductions used in Figure 4.16, and the proof file in Figure 4.17.

```
Primitive PRG(Int lambda, Int stretch) {
    Int lambda = lambda;
    Int stretch = stretch;
    BitString<lambda + stretch> Evaluate(BitString<lambda> x);
}
```

Figure 4.12: ProofFrog syntax for a PRG primitive.

```
Scheme TriplingPRG(PRG G) extends PRG {
    requires G.lambda == G.stretch;
    Int lambda = G.lambda;
    Int stretch = 2 * G.lambda;
    BitString<lambda + stretch> Evaluate(BitString<lambda> s) {
        BitString<2 * lambda> result1 = G.Evaluate(s);
        BitString<lambda> x = result1[0 : lambda];
        BitString<lambda> y = result1[lambda : 2*lambda];
        BitString<2 * lambda> result2 = G.Evaluate(y);
        return x || result2;
    }
}
```

Figure 4.13: ProofFrog syntax for a length-tripling PRG scheme.

```
Game Real(PRG G) {
    BitString<G.lambda + G.stretch> Query() {
        BitString<G.lambda> s <- BitString<G.lambda>;
        return G.Evaluate(s);
    }
}
Game Random(PRG G) {
    BitString<G.lambda + G.stretch> Query() {
        BitString<G.lambda + G.stretch> r <- BitString<G.lambda + G.stretch>;
        return r;
    }
}
export as IND; // Short for indistinguishable
```

Figure 4.14: ProofFrog syntax for the pair of games modelling PRG security.

```
Game Concatenate(Int len1, Int len2) {
    BitString<len1 + len2> Query() {
        BitString<len1> x <- BitString<len1>;
        BitString<len2> y <- BitString<len2>;
        return x || y;
    }
}
Game SampleDirectly(Int len1, Int len2) {
    BitString<len1 + len2> Query() {
        BitString<len1 + len2> value <- BitString<len1 + len2>;
        return value;
    }
}
export as BitStringSampling;
```

Figure 4.15: ProofFrog syntax for a pair of games modelling interchangeability between two methods of sampling bitstrings.

```
Reduction R1(PRG G, TriplingPRG T)
compose IND(G) against IND(T).Adversary {
    BitString<T.lambda + T.stretch> Query() {
        BitString<2 * T.lambda> result1 = challenger.Query();
        BitString<T.lambda> x = result1[0 : T.lambda];
        BitString<T.lambda> y = result1[lambda : 2*T.lambda];
        BitString<2 * T.lambda> result2 = G.evaluate(y);
        return x || result2;
    }
}
Reduction R2(TriplingPRG T)
compose BitStringSampling(T.lambda, T.lambda) against IND(T).Adversary {
    BitString<T.lambda + T.stretch> Query() {
        BitString<2 * T.lambda> result1 = challenger.Query();
        BitString<T.lambda> x = result1[0 : T.lambda];
        BitString<T.lambda> y = result1[lambda : 2*T.lambda];
        BitString<2 * T.lambda> result2 = G.evaluate(y);
        return x || result2;
    }
}
Reduction R3(PRG G, TriplingPRG T)
compose IND(G) against IND(T).Adversary {
    BitString<T.lambda + T.stretch> Query() {
        BitString<T.lambda> x <- BitString<lambda>;
        BitString<2 * T.lambda> result2 = challenger.Query();
        return x || result2;
    }
}
Reduction R4(TriplingPRG T)
compose BitStringSampling(T.lambda, 2 * T.lambda) against IND(T).Adversary {
    BitString<T.lambda + T.stretch> Query() {
        return challenger.Query();
    }
}
```

Figure 4.16: Reductions used to prove Theorem 3.

```
proof:
let:
    Int lambda;
    PRG G = PRG(lambda, lambda);
    TriplingPRG T = TriplingPRG(G);
assume:
    IND(G);
    BitStringSampling(lambda, lambda);
    BitStringSampling(lambda, 2 * lambda);
theorem:
    IND(T);
games:
    IND(T).Real against IND(T).Adversary; // Game 1
    IND(G).Real compose R1(G, T) against IND(T).Adversary; // Game 2
    IND(G).Random compose R1(G, T) against IND(T).Adversary; // Game 3
    BitStringSampling(lambda, lambda).SampleDirectly
        compose R2(T, lambda) against IND(T).Adversary; // Game 4
    BitStringSampling(lambda, lambda).Concatenate
        compose R2(T, lambda) against IND(T).Adversary; // Game 5
    IND(G).Real compose R3(G, T) against IND(T).Adversary; // Game 6
    IND(G).Random compose R3(G, T) against IND(T).Adversary; // Game 7
    BitStringSampling(lambda, 2 * lambda).Concatenate
        compose R4(T) against IND(T).Adversary; // Game 8
    BitStringSampling(lambda, 2 * lambda).SampleDirectly
        compose R4(T) against IND(T).Adversary; // Game 9
    IND(T).Random against IND(T).Adversary; // Game 10
```

Figure 4.17: Proof file for Theorem 3.

4.3.4 Simplifying Slices

The proof's first four hops are actually able to verify just using the previous strategies of inlining, instantiation, copy propagation, statement reordering, and variable renaming. However, on hop five, ProofFrog cannot verify the jump from `BitStringSampling(lambda, lambda).Concatenate compose R2(T, lambda)` to `IND(G).Real compose R3(G, T)`. The second AST corresponding to the `IND(G).Real compose R3(G, T)` game simplifies to:

```
Game Inlined() {
  BitString<lambda + 2 * lambda> Query() {
    BitString<lambda> x <- BitString<lambda>;
    BitString<lambda> challenger.Query@s <- BitString<lambda>;
    BitString<2 * lambda> result2 = G.evaluate(challenger.Query@s);
    return x || result2;
  }
}
```

Aside from variable renaming, this game AST corresponds exactly to $\mathbf{G}_2^H$ as written in the pen-and-paper proof. But, the first game AST, for `BitStringSampling(lambda, lambda).Concatenate compose R2(T, lambda)`, is only able to simplify to:

```
Game Inlined() {
  BitString<lambda + 2 * lambda> Query() {
    BitString<lambda> challenger.Query@x <- BitString<lambda>;
    BitString<lambda> challenger.Query@y <- BitString<lambda>;
    BitString<2 * lambda> result1 = challenger.Query@x || challenger.Query@y;
    BitString<lambda> x = result1[0 : lambda];
    BitString<lambda> y = result1[lambda : 2 * lambda];
    BitString<2 * lambda> result2 = G.evaluate(y);
    return x || result2;
  }
}
```

One can clearly see that the values `x` and `y` are copies of `challenger.Query@x` and `challenger.Query@y`. `result1` is just a concatenation of the two variables together, followed by the declarations of `x` and `y` extracting exactly those values which were previously concatenated. To transform the first AST into the second AST, we essentially require a more advanced form of copy propagation that can handle intermediate concatenations.

To achieve this, the `SimplifySliceTransformer` was added to the list of standard manipulations applied to all ASTs. It 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 the `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`. In the case of the hop in question: this transformation leads to `x` being assigned directly to `challenger.Query@x`, and `y` assigned directly to `challenger.Query@y`. After performing such a transformation, `result1` is no longer used; the variable declaration is considered dead code and is eliminated during statement re-ordering. The variables `x` and `y` are eliminated via a copy propagation pass. The ASTs are then identical up to variable renaming. The `SimplifySpliceTransformer`, in combination with the previously described transformations, suffices to prove this hop.

4.3.5 Symbolic Computation

ProofFrog also faces another stumbling block when attempting to hop from game 7, `IND(G).Random compose R3(G, T)`, to game 8, `BitStringSampling(lambda, 2 * lambda).Concatenate compose R4(T)`. This step is essentially just trying to codify the transition from $\mathbf{G}_3^H$ to $\mathbf{PRG}_{\text{rand}}^H$: sampling a bitstring of length λ and concatenating it with a bitstring of length 2λ is equivalent to sampling a bitstring of length 3λ. This step does not verify because the ASTs are not equivalent after transformation. The value `r` used in `G`'s `Query` method has type `BitString<G.lambda + G.stretch>`, which after instantiation becomes `BitString<lambda + lambda>`. The corresponding value `y` in the `BitStringSampling Query` method has type `BitString<len2>`, which becomes `BitString<2 * lambda>` after concatenation (recall from Figure 4.15 that `len2` is one of the parameters to the `BitStringSampling` game). As a result, the game canonicalization procedure fails; everything else is identical between the two games, the only difference is in the way the type parameterization is written. This is, strictly speaking, just a result of the way the proof was written. In fact, the proof can be verified with just a few changes on the user side: rewrite the `stretch` field in the `TriplingPRG` scheme definition as `G.lambda + G.lambda`, and rewrite the assumption involving the `BitStringSampling` game to have the parameter `lambda + lambda` instead of `2 * lambda`. Making these changes results in the proof succeeding where it did not before. The fact that the proof failed did not indicate that it was invalid, merely that ProofFrog did not have the capability to verify it. However, having to cater the syntax of one's proof to such exact specifications makes for an arduous user experience. It is better for ProofFrog to be able to recognize that `BitString<lambda + lambda>` and `BitString<2 * lambda>` represent the same type and subsequently verify the proof, rather than forcing the user to make changes to definitions that are already equivalent. To ensure this was the case, we implemented elements of symbolic computation using the SymPy library.

SymPy is a library developed for symbolic computation and computer algebra in Python [16]. Although it is a powerful piece of software covering many branches of mathematics, in ProofFrog, SymPy is used exclusively for simplifying mathematical expressions. The `SymbolicComputationTransformer` has the role of utilizing SymPy to simplify eligible expressions in ProofFrog. 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 of type `2 * lambda`. Variables with differing names will produce differing symbols which SymPy will not collapse. If an eligible expression can be simplified, we then manually convert the SymPy AST into the equivalent ProofFrog AST, and replace the expression node with a simplified node. This transformation allows ProofFrog to simplify the type parameterization for both games in the hop to `BitString<2 * lambda>` instead of `BitString<lambda + lambda>`. Since this was the only difference between the two ASTs, the hop, as well as the rest of the proof, can now be verified. In Figure 4.18 we present a new flowchart for ProofFrog's functionality, with the added transformations included.

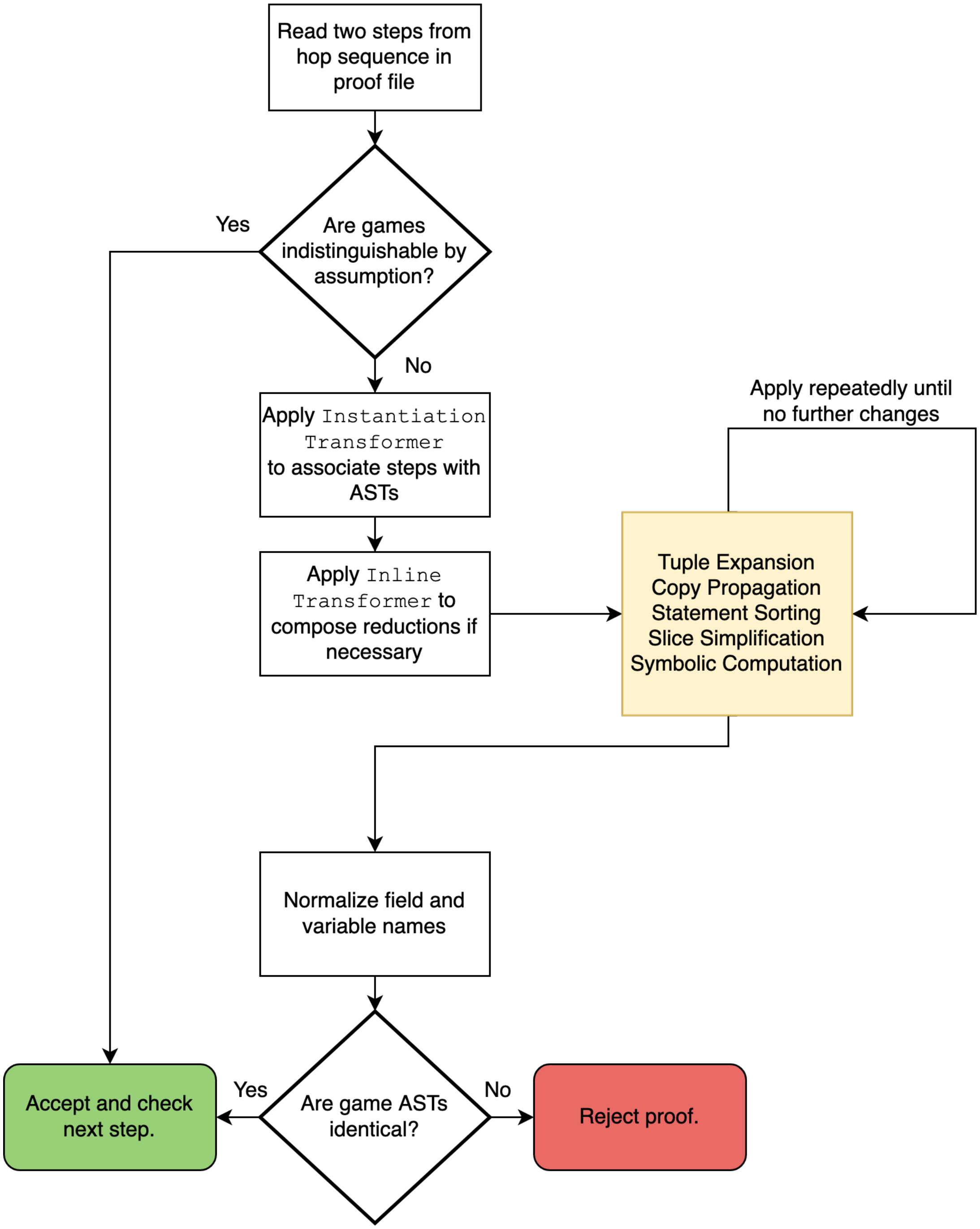
Read two steps from hop sequence in proof file
Are games indistinguishable by assumption?
Yes
No
Apply Instantiation Transformer to associate steps with ASTs
Apply Inline Transformer to compose reductions if necessary
Apply repeatedly until no further changes
Tuple Expansion
Copy Propagation
Statement Sorting
Slice Simplification
Symbolic Computation
Normalize field and variable names
Are game ASTs identical?
Yes
No
Accept and check next step.
Reject proof.

Figure 4.18: A flowchart of ProofFrog engine functionality necessary to prove Theorem 3.

4.4 One-time Secrecy Implies CPA Security for Public-Key Encryption Schemes

All of the previous proofs have utilized the game-hopping technique with a constant number of games. However, some cryptographic proofs require use of a “hybrid argument”, in which the number of games may rely on the adversary’s interaction with the challenger. This section will detail a proof for a property of public-key encryption schemes that utilizes a hybrid argument. In doing so we will also explore many other ProofFrog features that were developed for transforming more complex programs. These features include simplifications applied to branches, allowing the user to specify assumptions that bound possible values of fields, integration with the Z3 theorem prover, and recognizing when fields are unnecessary or identical.

4.4.1 Definitions

Some previous examples utilized symmetric encryption schemes. Public-key encryption schemes are another often studied cryptographic construction. In contrast to symmetric encryption schemes, public-key encryption schemes do not require the communicating parties to establish a shared secret prior to communication.

Definition 11 (Definition 15.0 from [18])**.** *A public-key encryption scheme consists of a public key space* P*, a secret key space* S*, a message space* M*, a ciphertext space* C*, and the following algorithms:*

- KeyGen*: a randomized algorithm that outputs a pair* $(pk, sk) \in \mathrm{P} \times \mathrm{S}$
- Enc*: a (possibly randomized) algorithm that takes a public key* $pk \in P$ *and a plaintext* $m \in \mathrm{M}$ *as input, and outputs a ciphertext* $c \in \mathrm{C}$*.*
- Dec*: a deterministic algorithm that takes a secret key* $sk \in S$ *and ciphertext* $c \in \mathrm{C}$ *as input, and outputs a plaintext* $m \in \mathrm{M}$*, or the symbol* None *to indicate failure to decrypt.*

Public-key encryption schemes assume that anyone can encrypt messages using the public key, but only the individual who possesses the secret key may decrypt ciphertexts. As a result, security definitions for public-key encryption schemes mirror those defined for symmetric encryption schemes, except that the adversary is provided the public key. For example, CPA security for public-key encryption schemes is defined identically apart from this one change.

Definition 12 (Definition 15.1 from [18])**.** *A public-key encryption scheme* Σ *is CPA secure if and only if* $\boldsymbol{CPA\text{-}PK}_L^\Sigma \approx \boldsymbol{CPA\text{-}PK}_R^\Sigma$*, where:*

```
$\mathbf{CPA\text{-}PK}_{\mathrm{L}}^{\Sigma}$

$(pk, sk) \leftarrow \Sigma.\mathrm{KeyGen}()$
Give $pk$ to adversary

$\mathrm{Eavesdrop}(m_L, m_R \in \Sigma.\mathrm{M})$:
    $c := \Sigma.\mathrm{Enc}(pk, m_L)$
    return $c$
```

```
$\mathbf{CPA\text{-}PK}_{\mathrm{R}}^{\Sigma}$

$(pk, sk) \leftarrow \Sigma.\mathrm{KeyGen}()$
Give $pk$ to adversary

$\mathrm{Eavesdrop}(m_L, m_R \in \Sigma.\mathrm{M})$:
    $c := \Sigma.\mathrm{Enc}(pk, m_R)$
    return $c$
```

Previously we modelled some different types of security definitions by changing the scoping of the key generation: for example, the one-time uniform ciphertexts property required regenerating the key with each oracle call so that the adversary could not observe relationships between ciphertexts encrypted with the same key. However, with public-key encryption schemes, the key cannot be regenerated when the oracle is called because it weakens the attack possibilities; to encompass all possible strategies, the adversary should be able to use the public key to encrypt as that may influence the messages they would like to pass to the Eavesdrop oracle. There is a solution which allows the adversary to use the public key before calling the oracle while also ensuring that the adversary cannot observe relationships between ciphertexts encrypted with the same key: simply generate the key during initialization, and return nothing if the adversary attempts to use the oracle more than once. An example is given in the form of the one-time secrecy definition, which is like CPA security except that an adversary may only observe one ciphertext.

Definition 13 (Definition 15.4 from [18])**.** *A public-key encryption scheme* Σ *has one-time secrecy if and only if* $\boldsymbol{OTS\text{-}PK}_L^\Sigma \approx \boldsymbol{OTS\text{-}PK}_R^\Sigma$*, where:*

```
$\textbf{OTS-PK}^{\Sigma}_{\text{L}}$

count := 0
$(pk, sk) \leftarrow \Sigma.\text{KeyGen}()$
Give $pk$ to adversary

Eavesdrop($m_L, m_R \in \Sigma.\text{M}$):
    count := count + 1
    if count > 1: return None
    $c := \Sigma.\text{Enc}(pk, m_L)$
    return $c$
```

```
$\textbf{OTS-PK}^{\Sigma}_{\text{R}}$

count := 0
$(pk, sk) \leftarrow \Sigma.\text{KeyGen}()$
Give $pk$ to adversary

Eavesdrop($m_L, m_R \in \Sigma.\text{M}$):
    count := count + 1
    if count > 1: return None
    $c := \Sigma.\text{Enc}(pk, m_R)$
    return $c$
```

4.4.2 Theorem and Proof

The following theorem relates the previous two security definitions. It also marks an interesting contrast in that the result applies only for public-key encryption schemes and not for symmetric encryption schemes.

Theorem 4 (Claim 15.5 from [18])**.** *Let Σ be a public-key encryption scheme. If Σ has one-time secrecy, then Σ is CPA secure.*

Proof. As previously mentioned, this proof uses a hybrid argument. The number of games in the proof will depend on the number of calls that the adversary makes to the Eavesdrop oracle, which we will denote as q. Because the adversary must run in polynomial time with respect to the security parameter, q is also polynomial with respect to the security parameter. For each $i \in \{1, 2, ..., q\}$, define the following reduction $\mathbf{R}_i^{\Sigma}$:

```
$\mathbf{R}_i^{\Sigma}$

Receive $pk$ from challenger
count := 0
Give $pk$ to adversary

Eavesdrop($m_L, m_R \in \Sigma$.M):
    count := count + 1
    if count $< i$:
        return $\Sigma$.Enc($pk, m_R$)
    else if count $\overset{?}{=} i$:
        return challenger.Eavesdrop($m_L, m_R$)
    else:
        return $\Sigma$.Enc($pk, m_L$)
```

Then, we can construct a sequence of games:

1. $\textbf{CPA-PK}_{\mathrm{L}}^{\Sigma} \equiv \textbf{OTS-PK}_{\mathrm{L}}^{\Sigma} \circ \mathbf{R}_1^{\Sigma}$, because in each version of the reduction game, count is always at least 1. In the first call, Eavesdrop will encrypt the left message via the call to $\textbf{OTS-PK}_{\mathrm{L}}^{\Sigma}$'s Eavesdrop oracle, and for all subsequent calls, the left message will be encrypted directly by the else branch.

2. For each $i \in \{1, 2, ..., q\}$: $\textbf{OTS-PK}_{\mathrm{L}}^{\Sigma} \circ \mathbf{R}_i^{\Sigma} \approx \textbf{OTS-PK}_{\mathrm{R}}^{\Sigma} \circ \mathbf{R}_i^{\Sigma}$, since Σ is assumed to have one-time secrecy.

3. For each $i \in \{1, 2, ..., q-1\}$: $\textbf{OTS-PK}_{\mathrm{R}}^{\Sigma} \circ \mathbf{R}_i^{\Sigma} \equiv \textbf{OTS-PK}_{\mathrm{L}}^{\Sigma} \circ \mathbf{R}_{i+1}^{\Sigma}$ because the first i calls encrypt the right message, and the remaining calls encrypt the left message. The difference is just that in the first game, the i-th call encrypts the left message by calling the challenger oracle, whereas in the second game, the i-th call is encrypted directly by the reduction. And in the first game, the $(i+1)$-th call is handled by the else branch encrypting the left message, whereas in the second game, the $(i+1)$-th call is handled by the challenger oracle encrypting the left message.

4. $\textbf{OTS-PK}_{\mathrm{R}}^{\Sigma} \circ \mathbf{R}_q^{\Sigma} \equiv \textbf{CPA-PK}_{\mathrm{R}}^{\Sigma}$, because count is always at most q, the oracle will encrypt the right message directly for the first $q-1$ calls, and encrypt it via the call to $\textbf{OTS-PK}_{\mathrm{R}}^{\Sigma}$'s Eavesdrop oracle for the q-th call.

Because q is polynomial in terms of the security parameter, we have applied indistinguishability of the one-time secrecy of Σ a polynomial number of times. The advantage of an adversary in distinguishing the one-time-secrecy games is negligible with respect to the security parameter. A polynomial quantity times a negligible quantity is still negligible, and so a negligible advantage has been built up by these q hops. Hence, $\textbf{CPA-PK}^{\Sigma}_{\mathrm{L}} \approx \textbf{CPA-PK}^{\Sigma}_{\mathrm{R}}$ and Σ is CPA secure. □

4.4.3 ProofFrog Encoding

To encode this proof requires a new primitive for public-key encryption schemes (Figure 4.19). We must also specify game pair definitions for both one-time secrecy (Figure 4.20) and CPA security (Figure 4.21). The definition of one-time secrecy utilizes an optional type modifier. Given a type `T`, a value of type `T?` is either of type `T` or is the special symbol `None`. The `None` value is used in the one-time secrecy game as a return value if the adversary calls the `Eavesdrop` oracle more than once. Finally, we specify the reduction used in the hybrid argument (Figure 4.22), and the proof steps (Figure 4.23). The proof steps have new syntax in the form of an `induction` block (used for the variable hybrid proof), and some `assume` statements. The functionality of these new pieces of syntax is explained in full detail in later subsections.

```
Primitive PubKeyEnc(Set MessageSpace, Set CiphertextSpace,
Set PKeySpace, Set SKeySpace) {
    Set Message = MessageSpace;
    Set Ciphertext = CiphertextSpace;
    Set PublicKey = PKeySpace;
    Set SecretKey = SKeySpace;
    PublicKey * SecretKey KeyGen();
    Ciphertext Enc(PublicKey pk, Message m);
    Message Dec(SecretKey sk, Message m);
}
```

Figure 4.19: ProofFrog syntax for a public-key encryption scheme.

```
Game Left(PubKeyEnc E) {
    E.PublicKey pk;
    Int count;
    E.PublicKey Initialize() {
        E.PublicKey * E.SecretKey k = E.KeyGen();
        pk = k[0];
        count = 0;
        return pk;
    }
    E.Ciphertext? Eavesdrop(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;
    Int count;
    E.PublicKey Initialize() {
        E.PublicKey * E.SecretKey k = E.KeyGen();
        pk = k[0];
        count = 0;
        return pk;
    }
    E.Ciphertext? Eavesdrop(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 OTS;
```

Figure 4.20: ProofFrog syntax for the pair of games modelling one-time secrecy for public-key encryption schemes.

```
Game Left(PubKeyEnc E) {
    E.PublicKey pk;
    E.PublicKey Initialize() {
        E.PublicKey * E.SecretKey k = E.KeyGen();
        pk = k[0];
        return pk;
    }
    E.Ciphertext Eavesdrop(E.Message mL, E.Message mR) {
        return E.Enc(pk, mL);
    }
}
Game Right(PubKeyEnc E) {
    E.PublicKey pk;
    E.PublicKey Initialize() {
        E.PublicKey * E.SecretKey k = E.KeyGen();
        pk = k[0];
        return pk;
    }
    E.Ciphertext Eavesdrop(E.Message mL, E.Message mR) {
        return E.Enc(pk, mR);
    }
}
export as CPA;
```

Figure 4.21: ProofFrog syntax for the pair of games modelling CPA security for public-key encryption schemes.

```
Reduction R(PubKeyEnc E, Int i) compose OTS(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 Eavesdrop(E.Message mL, E.Message mR) {
        count = count + 1;
        if (count < i) {
            return E.Enc(pk, mR);
        } else if (count == i) {
            return challenger.Eavesdrop(mL, mR);
        } else {
            return E.Enc(pk, mL);
        }
    }
}
```

Figure 4.22: Reduction used in the proof of Theorem 4.

```
proof:
let:
    Set MessageSpace;
    Set CiphertextSpace;
    Set PubKeySpace;
    Set SecretKeySpace;
    Int q;
    PubKeyEnc E = PubKeyEnc(
        MessageSpace, CiphertextSpace, PubKeySpace, SecretKeySpace
    );
assume:
    OTS(E);
theorem:
    CPA(E);
games:
    CPA(E).Left against CPA(E).Adversary;
    assume R(E, 1).count >= 1;
    assume OTS(E).Left.count == 1;
    induction(i from 1 to q) {
        OTS(E).Left compose R(E, i) against CPA(E).Adversary;
        OTS(E).Right compose R(E, i) against CPA(E).Adversary;
        assume OTS(E).Left.count == 1;
        assume OTS(E).Right.count == 1;
    }
    assume R(E, q).count < q + 1;
    assume OTS(E).Right.count == 1;
    CPA(E).Right against CPA(E).Adversary;
```

Figure 4.23: Proof file for Theorem 4.

4.4.4 Induction

In prior examples, ProofFrog would simply check each pair of games listed in the sequence for interchangeability or indistinguishability. This approach is no longer possible with the introduction of the `induction` block, utilized for proving hybrid arguments. In order to verify the hybrid argument, ProofFrog will take a number of steps whenever it encounters an `induction` block. First, it will verify the base case of the induction. To do so, Proof-

Frog will create an AST corresponding to the first game in the block where the induction variable (`i`, in this case) is substituted with its starting value (1, in this case). This AST, after applying the standard canonicalization procedures, must be interchangeable with the AST created from the game listed immediately prior to the induction. Second, ProofFrog will check that each pair of games listed inside the `induction` block are indistinguishable or interchangeable. For these checks, the induction variable is left untouched when instantiating the games, since the hop should verify for a general `i` value. Third, ProofFrog will check the inductive step: it will ensure that the last game in the block is interchangeable with the first game in the block instantiated with `i + 1` for the induction variable. 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 (`q`, in this case). This AST must be interchangeable with the AST created from the game listed immediately after the induction. Verifying each of these hops is what allows ProofFrog to ensure the correctness of a hybrid argument.

4.4.5 Duplicated Fields

In addition to the new hybrid argument, this proof also uses the ability for the challenger to pass some information to the adversary via a return statement in the `Initialize` method. In this case, it is the public key which is provided to the adversary. To illustrate why this poses a problem, see the AST for the inlined game corresponding to the induction base case:

```
Game Inlined() {
  PubKeySpace challenger@pk;
  Int challenger@count;
  Int count;
  PubKeySpace pk;
  PubKeySpace Initialize() {
    PubKeySpace * SecretKeySpace challenger.Initialize@k = E.KeyGen();
    challenger@pk = challenger.Initialize@k[0];
    challenger@count = 0;
    pk = challenger@pk;
    count = 0;
    return pk;
  }
  CiphertextSpace Eavesdrop(MessageSpace mL, MessageSpace mR) {
```

```
15      count = count + 1;
16      if (count < 1) {
17        return E.Enc(pk, mR);
18      } else if (count == 1) {
19        CiphertextSpace? challenger.Eavesdrop@result = None;
20        challenger@count = challenger@count + 1;
21        if (challenger@count == 1) {
22          challenger.Eavesdrop@result = E.Enc(challenger@pk, mL);
23        }
24        return challenger.Eavesdrop@result;
25      } else {
26        return E.Enc(pk, mL);
27      }
28    }
29  }
```

The `Initialize` method has been inlined and now contains the initialization for both the challenger and the reduction. The `pk` value which was initially set to the argument `one_time_pk` is now instead set directly to the `challenger@pk` value which was returned in the challenger's `Initialize` method. From observation, one can see that `pk` and `challenger@pk` are identical values: they are both set to the same value, and never changed. However, the inlined game references both: it uses `pk` to encrypt in lines 18 and 27, whereas it uses `challenger@pk` to encrypt in line 23 (this is as a result of this statement having been inlined from the `challenger.Eavesdrop` oracle call). The duplicate fields will cause a problem with canonicalization, as `CPA(E).Left` only uses a single `pk` field to perform all of its encryption. As such, ProofFrog requires the ability to detect and unify duplicated fields: those which maintain the same value throughout the entire game's lifetime.

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. Applying this transformation to the inlined game causes all uses of `pk` to be rewritten in terms of `challenger@pk`, which brings the games one step closer to canonicalization.

4.4.6 Assumptions with Z3

The primary issue remaining that prevents the first two game ASTs from being equated is the vast difference in the two `Eavesdrop` methods. `CPA(E).Left` has a one-line `Eavesdrop` method, which just returns the encryption of the left message. `OTS(E).Left compose R(E, 1)`, which was shown above, has many branches concerning the values of the `count` and `challenger@count` fields. One can observe that `count` will always be at least one and hence the first branch, encrypting the right message, is dead code. Furthermore, since `count` monotonically increases, `count == 1` will only ever be triggered for one call of the oracle, which implies that in all cases, `challenger@count == 1`, and the second branch simplifies to just returning the encryption of the left message. Since both relevant branches just return the encryption of the left message, the behaviour is the same as that of `CPA(E).Left`. However, statically determining the possible values that fields may take on throughout the game's lifetime is challenging, in great part due to the undecidability of the problem. Rather than attempting to hard-code recognition of such cases, we instead took the approach of allowing proof authors to specify assumptions between hops that could be used during canonicalization. For this hop, it takes the form of the statements `assume R(E, 1).count >= 1` and `assume OTS(E).Left.count == 1`. These assumptions state the true observations one can make to ascertain that the behaviours of the two oracles are the same. 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.

To actually leverage these stated assumptions, ProofFrog delegates to Z3. Z3 is a satisfiability modulo theories (SMT) solver used extensively for static analysis of programs and can in many cases determine whether logical formulae are satisfiable or unsatisfiable [10]. The assumption simplification process begins by taking each assumption statement listed and converting it into a Z3 formula. Type checking is performed to ensure that only constructs that Z3 supports, like booleans and integers, are encoded. Then, the transformer will search the program for any operation using an operator in the set {`==, !=, <=, <, >, >=, &&, ||, !`}. If this operation is identical to a provided assumption, then the value can be directly replaced with `true`. For example, we specify in the proof the assumption that `OTS(E).Left.count == 1`. The variable `OTS(E).Left.count` is transformed when inlining into the variable `challenger@count`. Then, when the transformer encounters the condition `challenger@count == 1`, by user assumption it will be replaced with `true`. If an operation o is not exactly identical with the user-stated assumption a, then ProofFrog will use Z3.

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 operation o by using an assumption a. First, if $a \implies o$ is a tautology, then since a is assumed to be true, we must have that o is true as well. Second, if $a \wedge o$ is a contradiction (unsatisfiable, in Z3 parlance) then since a is assumed to be true, we must have that o is false. Since Z3 does not have the ability to evaluate tautologies, we can instead use Z3 to evaluate $\neg(a \implies o)$. If this formula is unsatisfiable, we can directly replace o with `true`. Otherwise, we can evaluate $a \wedge o$, and replace o with `false` if Z3 deems the formula unsatisfiable. The use of Z3 allows us to replace some conditionals directly with boolean values. In the particular case of this proof, it allows us to transform statements like `challenger@count == 1` into `true` and `count < 1` to `false`, which will then be further simplified in later passes.

4.4.7 Branch Elimination

After handling duplicated fields and applying the assumptions from Z3, the `Eavesdrop` oracle for `OTS(E).Left compose R(E, 1)` has been simplified to the following:

```
CiphertextSpace Eavesdrop(MessageSpace mL, MessageSpace mR) {
  count = count + 1;
  if (false) {
    return E.Enc(challenger@pk, mR);
  } else if (count == 1) {
    CiphertextSpace? challenger.Eavesdrop@result = None;
    challenger@count = challenger@count + 1;
    if (true) {
      challenger.Eavesdrop@result = E.Enc(challenger@pk, mL);
    }
    return challenger.Eavesdrop@result;
  } else {
    return E.Enc(challenger@pk, mL);
  }
}
```

At this point, the `BranchEliminationTransformer` has the task of taking if-statements where the truth values of the conditions are explicitly known, and simplifying them. For example:

- A branch where the condition is `false` can have that condition and associated block removed. If this is the only condition, the if-statement can be removed entirely.
- A branch where the condition is `true` can have 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 can be replaced with the contents of the `else` block.

Performing these transformations simplifies the `Eavesdrop` method further to:

```
CiphertextSpace Eavesdrop(MessageSpace mL, MessageSpace mR) {
  count = count + 1;
  if (count == 1) {
    CiphertextSpace? challenger.Eavesdrop@result = None;
    challenger@count = challenger@count + 1;
    challenger.Eavesdrop@result = E.Enc(challenger@pk, mL);
    return challenger.Eavesdrop@result;
  } else {
    return E.Enc(challenger@pk, mL);
  }
}
```

4.4.8 Unnecessary Fields

We now note that the simplifications applied so far have made `challenger@count` a redundant field. It is set to 0 initially and incremented in one oracle call, but its value cannot affect the value of any oracle's return statement. As a result, we can remove the field and any statements that reference the field without changing the game's overall behaviour. The `UnnecessaryFieldVisitor` and `RemoveFieldTransformer` are what achieve this behaviour. The purpose of the `UnnecessaryFieldVisitor` is to return a list of all fields which do not affect the behaviour of any oracle. To do so, it performs a traversal of the dependency graph as previously described when sorting statements. The generation of the dependency graph is modified slightly for this purpose. Previously, return statements depended on all prior statements that modified fields, because even if a field did not affect the value of a return statement in one oracle, removing a field modification may change the return value of another oracle. In this case, however, we are not just sorting the statements of each oracle on an individual basis, rather, we are attempting to determine if a field does not affect the return value of any oracle. As such, we only want to consider a dependency between a field and a return statement if the value of the return statement depends on the value of the field. If a field does not occur in the traversal of the dependency graph for all return statements in all oracles, then that field is deemed unnecessary. The `RemoveFieldTransformer`, when given a list of fields, will remove each field from the game's AST, and 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. Performing these transformations is enough to remove the `challenger@count` field and bring the two ASTs one step closer to canonicalization. At this point, the `count` field is still considered necessary because it determines which branch is returned from in the `Eavesdrop` method. But, after a few further simplifications, it too will be removed to achieve equality between the two ASTs.

4.4.9 Return Canonicalization

Having removed the `challenger@count` variable, the `Eavesdrop` method now has the following body:

```
count = count + 1;
if (count == 1) {
  CiphertextSpace? challenger.Eavesdrop@result = None;
  challenger.Eavesdrop@result = E.Enc(challenger@pk, mL);
  return challenger.Eavesdrop@result;
} else {
  return E.Enc(challenger@pk, mL);
}
```

Both of the branches now clearly return `E.Enc(challenger@pk, mL)`, but the first branch does so with a local variable that is overwritten, and the second just returns the value of the function call directly. To canonicalize these to the same format, we simply apply a transformation that prioritizes returning expressions over returning variables. That is, if a return statement returns a variable, the transformer will scan previous statements to find the last time that variable was assigned. If the value can be statically determined (e.g, if the variable was not conditionally modified), we remove the last statement that assigns to the variable, and instead return that value directly. In this case, it allows us to return `E.Enc(challenger@pk, mL)` in the first block, and the line which sets `challenger.Eavesdrop@result = None` is removed later in a dead-code elimination pass. This results in both blocks having the exact same code: `return E.Enc(challenger@pk, mL);`

4.4.10 Branch Collapsing

Now that both blocks have the exact same code, we can discuss simplifications that can be applied to duplicated blocks of code. If an if-statement has multiple conditions that execute the same block of code, we can perform some transformations to the statement to ensure that a repeated block of code only appears once in the AST. This is achieved in ProofFrog via the `SimplifyIfTransformer`, which applies the following 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 meaning of the program. Assuming that the conditions do not contain function calls, then the first transformation preserves semantics because the block will be executed if either of the two conditions are satisfied. The second transformation preserves semantics because in both cases, the block will be executed if all prior conditions are evaluated to be `false`. Finally, the third transformation preserves semantics because in either representation, the if-statement will always execute the block `b`. Prior transformations yielded the following if-statement in the `Eavesdrop` method:

```
count = count + 1;
if (count == 1) {
  return E.Enc(challenger@pk, mL);
} else {
  return E.Enc(challenger@pk, mL);
}
```

When the `SimplifyIfTransformer` is applied to this if-statement, it collapses the first block into the `else` block, and then replaces the entire statement with `if (true)` { `return E.Enc(chllanerge@pk, mL);` }. It then simply takes two further passes for ProofFrog to verify this hop: first, the `BranchEliminationTransformer` will remove the `if (true)` condition and replace it with just the return statement. And then, the `UnnecessaryFieldVisitor` will identify that `count` has no effect on the return value of any oracle, and as such can be removed. This leaves both `CPA(E).Left` and `OTS(E).Left compose R(E, 1)` with the same AST up to variable renaming:

```
Game Inlined() {
  PubKeySpace challenger@pk;
  PubKeySpace Initialize() {
    PubKeySpace * SecretKeySpace challenger.Initialize@k = E.KeyGen();
    challenger@pk = challenger.Initialize@k[0];
    return pk;
  }
  CiphertextSpace Eavesdrop(MessageSpace mL, MessageSpace mR) {
    return E.Enc(pk, mL);
  }
}
```

All of these transformations, along with those described in previous sections, suffice to prove the first hop. As previously mentioned, this proof consists of four steps: the starting case for the induction, the hops inside the induction, the inductive step showing that $i \implies i+1$, and the ending case. The hop inside the induction is `OTS(E).Left compose R(E, i)` to `OTS(E).Right compose R(E, i)`, which is valid by our indistinguishability assumption for the `OTS` game. The transformations described so far also suffice to prove the ending step of the induction step: that `OTS(E).Right compose R(E, q)` is interchangeable with `CPA(E).Right`. However, there is one last sticking point with the inductive step that requires another proof engine feature to verify.

4.4.11 Condition Equivalence

To verify the inductive step, ProofFrog must establish interchangeability of `OTS(E).Right compose R(E, i)` and `OTS(E).Left compose R(E, i+1)`. These two games are interchangeable: in both of them, the first i calls encrypt the right message, and the remaining calls encrypt the left message. The difference is simply that in `OTS(E).right compose R(E, i)`, the i-th call is handled by the challenger, and encrypts the right message, whereas in `OTS(E).Left compose R(E, i+1)`, the i-th call is handled by the reduction encrypting the right message. These slight differences yield two different results after performing canonicalization. In the first AST, the block encrypting right messages is collapsed with the block making the challenger calls, yielding the following `Eavesdrop` method:

```
CiphertextSpace Eavesdrop(MessageSpace mL, MessageSpace mR) {
  count = count + 1;
  if (count < i || count == i) {
    return E.Enc(challenger@pk, mR);
  } else {
    return E.Enc(challenger@pk, mL);
  }
}
```

In the second AST, the challenger call encrypts the left message, and so is instead collapsed into the `else` block:

```
CiphertextSpace Eavesdrop(MessageSpace mL, MessageSpace mR) {
  count = count + 1;
  if (count < i + 1) {
    return E.Enc(challenger@pk, mR);
  } else {
    return E.Enc(challenger@pk, mL);
  }
}
```

Excluding the difference in the if-statement's condition, these ASTs are identical up to variable renaming. Since `i` is an integer, these conditions are semantically equivalent, but the hop will be rejected because the ASTs are not identical. Canonicalizing conditions is challenging; a set of just boolean conditions could be converted into a normal form, but that does not consider ordering of clauses which would still be necessary for an AST-level comparison. Furthermore, a normal form for booleans still does not handle cases like the `Eavesdrop` method where the same condition is written in two different ways. For if-statements, we therefore chose to pursue an alternative strategy beyond just AST comparison. Previously, if AST-level comparison failed, ProofFrog would simply reject the hop altogether. Now, instead, ProofFrog will first check if the ASTs are identical excluding conditions in if-statements. If there are further changes, the hop will be rejected as usual. However, if the changes are localized exclusively to the conditions in 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. As before, 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 extra logic surrounding equivalence of conditions is sufficient to verify the inductive step in this proof.

This section has described the verification of Theorem 4. In doing so, we introduced a number of new features to ProofFrog. In addition to adding a variety of new AST transformations to the proof engine, we also added broader functionality for verifying hops in the form of support for hybrid arguments via the `induction` block, and checking for conditional equivalence with Z3 when canonicalization fails to yield two equivalent ASTs. A new flowchart indicating all the functionality of ProofFrog up to this point is illustrated in Figure 4.24.

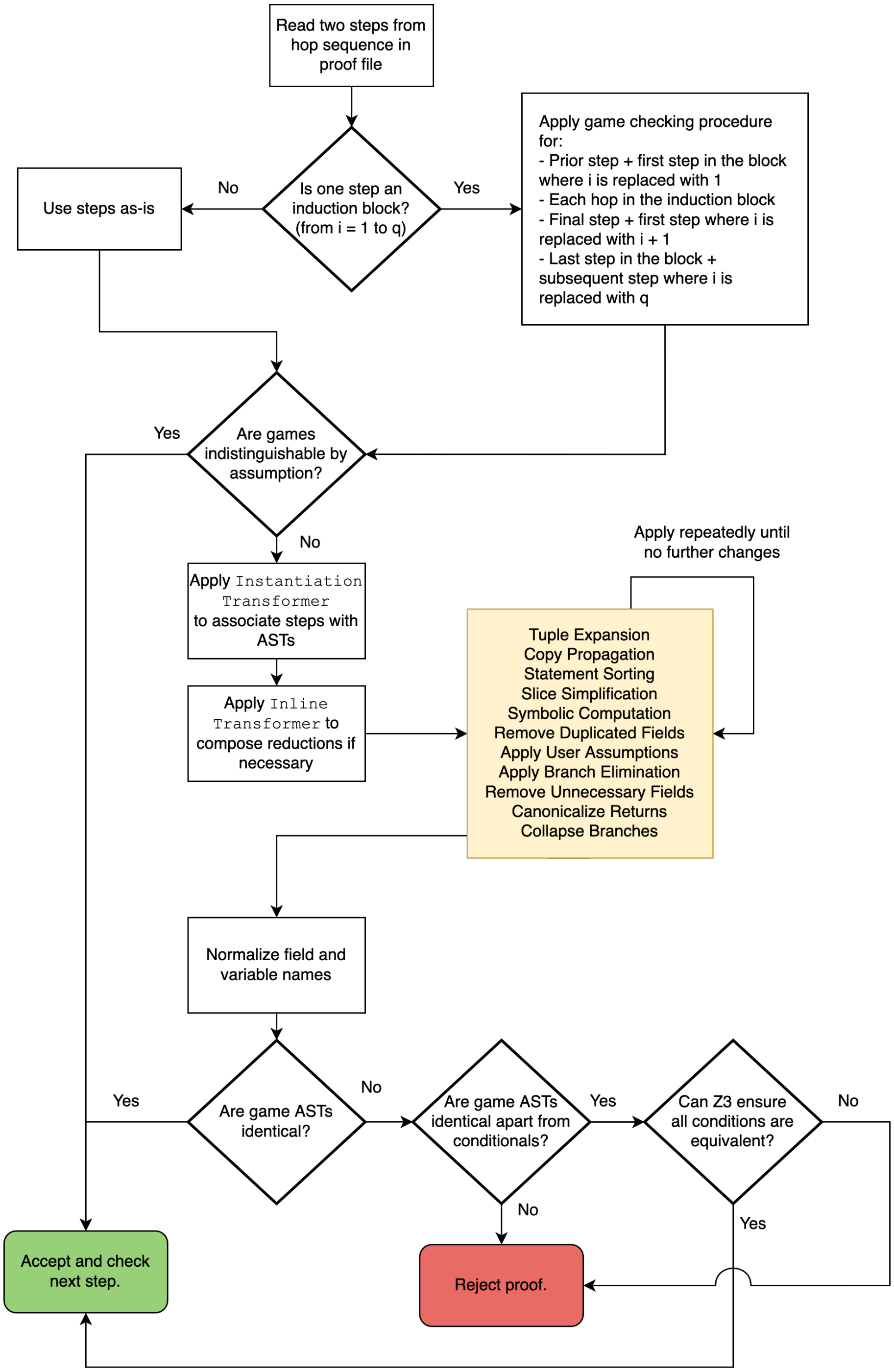
Read two steps from hop sequence in proof file
Is one step an induction block? (from i = 1 to q)
No
Yes
Use steps as-is
Apply game checking procedure for:
- Prior step + first step in the block where i is replaced with 1
- Each hop in the induction block
- Final step + first step where i is replaced with i + 1
- Last step in the block + subsequent step where i is replaced with q
Are games indistinguishable by assumption?
Yes
No
Apply Instantiation Transformer to associate steps with ASTs
Apply Inline Transformer to compose reductions if necessary
Apply repeatedly until no further changes
Tuple Expansion
Copy Propagation
Statement Sorting
Slice Simplification
Symbolic Computation
Remove Duplicated Fields
Apply User Assumptions
Apply Branch Elimination
Remove Unnecessary Fields
Canonicalize Returns
Collapse Branches
Normalize field and variable names
Are game ASTs identical?
Yes
No
Are game ASTs identical apart from conditionals?
Yes
No
Can Z3 ensure all conditions are equivalent?
No
Yes
Accept and check next step.
Reject proof.

Figure 4.24: A flowchart of ProofFrog engine functionality necessary to prove Theorem 4.

4.5 Encrypt-then-MAC is CCA Secure

Each proof presented in previous examples utilized reductions that applied to the same primitive. For example, for proofs involving symmetric or public-key encryption schemes, we used reductions to other symmetric or public-key encryption scheme security definitions. For the length-tripling PRG, we used reductions to the security of another PRG. The upcoming worked example deviates, in that it utilizes the security of one primitive to reason about the security of another. For this proof, we will also present a stronger notion of security for symmetric encryption schemes than previously seen before, as well as a new cryptographic primitive that can be used to achieve this stronger notion of security. This example requires some minor new additions to the proof engine to achieve verification, but beyond these additions, the proof can be verified mostly from the transformations previously described. In previous sections, the focus was primarily on the new features being introduced to the proof engine. This section, in contrast, aims to demonstrate how the transformations developed for ProofFrog so far are applicable to new proofs they were not explicitly designed for. In addition, this example applies reasoning across multiple oracles whereas all previous examples used solely one oracle.

4.5.1 Definitions

Previously, we described CPA security as a way to determine if an adversary can break some scheme (e.g, determine between left and right messages being encrypted) when given access to multiple ciphertexts encrypted with the same key. However, this is not the strongest adversary we can conceive of; namely, we can also consider an adversary that has the ability to decrypt arbitrary ciphertexts (excluding ciphertexts returned from the Eavesdrop oracle, as decrypting those would allow the adversary to immediately distinguish which game they are playing against). If the adversary still is not able to determine whether left messages or right messages are being encrypted when given access to this decryption oracle, then the scheme is called CCA secure (CCA standing for chosen ciphertext attack). Although in a real-world situation it is unlikely for an adversary to be able to decrypt any ciphertext (as it is assumed they do not possess the secret key), this definition also encompasses weaker attacks which have been utilized successfully to exploit non CCA secure schemes. The classic example is the Bleichenbacher attack against SSL, which utilizes server responses to determine whether a given ciphertext corresponds to a well-formed message, and from that information retrieves a secret key [8]. The games provided in the CCA security definition maintain a private secret key and a private set of ciphertexts that have been returned. They each provide two oracles to the adversary. One oracle is the Eavesdrop oracle, which accepts two messages m_L and m_R from the adversary, encrypts one of the messages, adds the corresponding ciphertext to its set of ciphertexts, and returns the ciphertext. The other is the Decrypt oracle, which accepts a ciphertext from the adversary, and returns the decrypted message so long as the ciphertext does not correspond to a message given to Eavesdrop. The two games in the CCA security definition only differ in that one always encrypts m_L, and the other always encrypts m_R.

Definition 14 (Definition 9.1 from [18])**.** *A symmetric encryption scheme* Σ *is CCA secure if and only if* $\boldsymbol{CCA}_L^{\Sigma} \approx \boldsymbol{CCA}_R^{\Sigma}$ *where:*

```
$\mathbf{CCA}_{\mathrm{L}}^{\Sigma}$

$k := \Sigma.\mathrm{KeyGen}()$
$\mathcal{S} := \emptyset$

Eavesdrop($m_L, m_R \in \Sigma.\mathrm{M}$):
    $c := \Sigma.\mathrm{Enc}(k, m_L)$
    $\mathcal{S} := \mathcal{S} \cup \{c\}$
    return $c$

Decrypt($c \in \Sigma.\mathrm{C}$):
    if $c \in \mathcal{S}$ return None
    return $\Sigma.\mathrm{Dec}(k, c)$
```

```
$\mathbf{CCA}_{\mathrm{R}}^{\Sigma}$

$k := \Sigma.\mathrm{KeyGen}()$
$\mathcal{S} := \emptyset$

Eavesdrop($m_L, m_R \in \Sigma.\mathrm{M}$):
    $c := \Sigma.\mathrm{Enc}(k, m_R)$
    $\mathcal{S} := \mathcal{S} \cup \{c\}$
    return $c$

Decrypt($c \in \Sigma.\mathrm{C}$):
    if $c \in \mathcal{S}$ return None
    return $\Sigma.\mathrm{Dec}(k, c)$
```

One way to achieve CCA security is to use a MAC, which is defined below.

Definition 15 (Definition 10.1 from [18])**.** *A message authentication code (MAC) scheme consists of a key space* $\mathcal{K}$*, a message space* $\mathcal{M}$*, a tag space* $\mathcal{T}$*, and the following algorithms:*

- KeyGen*: a randomized algorithm that outputs a key* $k \in \mathcal{K}$*.*
- MAC*: a deterministic algorithm that takes a key* k *and a message* $m \in \mathcal{M}$ *as input, and outputs a tag* t*.*

For a MAC to be worthwhile, it should be difficult to forge a tag for a message without access to the secret key. We can formalize this notion like usual, with a pair of security games. The real security game maintains a secret key for the MAC, and provides two oracles to the adversary: GetTag and CheckTag. GetTag accepts a message from the adversary and returns a corresponding tag computed from the message and the secret key. CheckTag accepts a message and a tag from the adversary and returns a boolean indicating whether the provided tag corresponds to the provided message. The random game also maintains a secret key for the MAC, as well as a set of message-tag pairs. Its GetTag oracle similarly returns a tag for a message but stores the message-tag pair in its set prior to returning. Its CheckTag oracle simply returns `true` if the message-tag pair has been previously returned from GetTag, and `false` otherwise.

Definition 16 (Definition 10.2 from [18])**.** *Let M be a MAC scheme. We say that M is* ***unforgeable*** *if* $\boldsymbol{MAC}^{M}_{real} \approx \boldsymbol{MAC}^{M}_{rand}$ *where:*

```
$\mathbf{MAC}^{M}_{\mathrm{real}}$

$k := M.\mathrm{KeyGen}()$

$\mathrm{GetTag}(m \in M.\mathcal{M})$:
    return $M.\mathrm{MAC}(k, m)$

$\mathrm{CheckTag}(m \in M.\mathcal{M}, t \in M.\mathcal{T})$:
    return $t \overset{?}{=} M.\mathrm{MAC}(k, m)$
```

```
$\mathbf{MAC}^{M}_{\mathrm{rand}}$

$k := M.\mathrm{KeyGen}()$
$\mathcal{S} := \emptyset$

$\mathrm{GetTag}(m \in M.\mathcal{M})$:
    $t := M.\mathrm{MAC}(k, m)$
    $\mathcal{S} := \mathcal{S} \cup \{(m, t)\}$
    return $t$

$\mathrm{CheckTag}(m \in M.\mathcal{M}, t \in M.\mathcal{T})$:
    return $(m, t) \overset{?}{\in} \mathcal{S}$
```

Recall that in this thesis, we model security definitions via pairs of games where the adversary's goal is to determine which game they are composed with. In a more traditional treatment of the material, the unforgeability security definition would be modelled by a challenger providing a GetTag oracle to the adversary, where the adversary wins if they can forge a valid tag for a message m without passing m to the GetTag oracle. Both the traditional unforgeability security definition and the unforgeability security definition based on indistinguishability are equivalent. Consider how an adversary could distinguish between the two games; the output distribution of the GetTag oracle is clearly identical in both. Therefore, in order to distinguish between the two games, an adversary would need to find a difference in the output of the CheckTag oracle, which can only be done by forging a tag for a message m without passing m to the GetTag oracle. The real game would return `true` for such a forgery, whereas the random game would return `false`, since the pair (m, t) would not appear in $\mathcal{S}$. Therefore, any strategy that an adversary can use to win the traditional unforgeability security definition can also be used to break indistinguishability, and vice versa; hence, the two definitions are equivalent.

Finally, we can construct a CCA secure symmetric encryption scheme by tagging each ciphertext that we produce with a MAC. If the MAC is secure, then this ensures that the decryption oracle yields no extra information, since an adversary cannot forge a valid tag for any ciphertext not returned from the Eavesdrop oracle.

Definition 17 (Construction 10.9 from [18]). *Let Σ be a symmetric encryption scheme and M a MAC scheme where $\Sigma.\mathrm{C} \subseteq M.\mathcal{M}$. **EtM (Encrypt-then-MAC)** is a symmetric encryption scheme where $\mathrm{K} = \Sigma.\mathrm{K} * M.\mathcal{K}$, $\mathrm{M} = \Sigma.\mathrm{M}$, $\mathrm{C} = \Sigma.\mathrm{C} * M.\mathcal{T}$ defined by:*

$$\mathrm{KeyGen}() = (\Sigma.\mathrm{KeyGen}(), M.\mathrm{KeyGen}())$$

$$\mathrm{Enc}((k_e, k_m), m) = (c := \Sigma.\mathrm{Enc}(k_e, m), t := M.\mathrm{MAC}(k_m, c))$$

$$\mathrm{Dec}((k_e, k_m), (c, t)) = \begin{cases} \Sigma.\mathrm{Dec}(k_e, c) & t = M.\mathrm{MAC}(k_m, c) \\ \mathrm{None} & \textit{otherwise} \end{cases}$$

4.5.2 Theorem and Proof

Theorem 5 (Claim 10.10 from [18]). *If Σ is CPA secure and M is a secure MAC, then EtM is CCA secure.*

For this proof, we will present an abridged version, where we only give the steps up to the replacement of the left message with the right message. All steps after this exchange occur exactly in reverse to the first half of the proof. As usual, we first provide an overview of the sequences of hops.

1. $\mathbf{CCA}_{\mathrm{L}}^{\mathrm{EtM}}$

- We rewrite the uses of the MAC scheme in the CCA game as calls to oracles of the real MAC game. This is an interchangeable hop.

2. $\mathbf{MAC}_{\mathrm{real}}^{M} \circ \mathbf{R}_{1}^{\mathrm{EtM}}$

- We exchange the real MAC unforgeability game with the random MAC unforgeability game, thanks to the assumed unforgeability of M. This is an indistinguishable hop.

3. $\mathbf{MAC}^{M}_{\text{rand}} \circ \mathbf{R}^{\text{EtM}}_{1}$

- We inline the reduction calls to form an intermediate game for explanation. This is an interchangeable hop.

4. $\mathbf{G}^{\text{EtM}}_{1}$

- We argue that two values are identical, and that some decryption code is unreachable. This is an interchangeable hop.

5. $\mathbf{G}^{\text{EtM}}_{2}$

- We write the encryption of the left message in terms of a call to the left CPA game. This is an interchangeable hop.

6. $\mathbf{CPA}^{\Sigma}_{L} \circ \mathbf{R}^{\text{EtM}}_{2}$

- We exchange the left CPA game with the right CPA game thanks to the assumed CPA security of Σ. This is an indistinguishable hop.

7. $\mathbf{CPA}^{\Sigma}_{R} \circ \mathbf{R}^{\text{EtM}}_{2}$

From this point, we proceed through the steps of the proof backwards, hopping to identical versions of $\mathbf{G}^{\text{EtM}}_{2}, \mathbf{G}^{\text{EtM}}_{1}, \mathbf{MAC}^{M}_{\text{rand}} \circ \mathbf{R}^{\text{EtM}}_{1}$ and $\mathbf{MAC}^{M}_{\text{real}} \circ \mathbf{R}^{\text{EtM}}_{1}$ except where m_L has been replaced with m_R. We finally conclude by hopping to $\mathbf{CCA}^{\text{EtM}}_{\text{R}}$.

To discuss each hop in more detail, we begin by showing $\mathbf{CCA}^{\text{EtM}}_{\text{L}}$ where the EtM definitions are inlined into the CCA game:

```
$\mathbf{CCA}_{\mathrm{L}}^{\mathrm{EtM}}$

$k_\Sigma := \Sigma.\mathrm{KeyGen}()$
$k_M := M.\mathrm{KeyGen}()$
$\mathcal{S} := \emptyset$

Eavesdrop($m_L, m_R \in \Sigma.\mathrm{M}$):
    $c := \Sigma.\mathrm{Enc}(k_\Sigma, m_L)$
    $t := M.\mathrm{MAC}(k_M, c)$
    $\mathcal{S} := \mathcal{S} \cup \{(c,t)\}$
    return $(c,t)$

Decrypt($(c,t) : c \in \Sigma.\mathrm{C},\ t \in M.\mathcal{T}$)
    if $(c,t) \in \mathcal{S}$ return None
    if $t \neq M.\mathrm{MAC}(k_M, c)$: return None
    return $\Sigma.\mathrm{Dec}(k_\Sigma, c)$
```

Next, we note that we can compute the value $M.\mathrm{MAC}(k_M, c)$ and check if $t \neq M.\mathrm{MAC}(k_M, c)$ by calling the CheckTag and GetTag oracles in the real MAC game. Hence, we write a reduction to do so, and use an interchangeable hop to demonstrate that $\mathbf{CCA}_{\mathrm{L}}^{\mathrm{EtM}} \equiv \mathbf{MAC}_{\mathrm{real}}^{M} \circ \mathbf{R}_1^{\mathrm{EtM}}$

```
$\mathbf{MAC}^{M}_{\text{real}}$

$k := M.\text{KeyGen}()$

GetTag($m \in M.\mathcal{M}$):
    return $M.\text{MAC}(k, m)$

CheckTag($m \in M.\mathcal{M}, t \in M.\mathcal{T}$):
    return $t \stackrel{?}{=} M.\text{MAC}(k, m)$
```

```
$\mathbf{R}^{\text{EtM}}_{1}$

$k_\Sigma := \Sigma.\text{KeyGen}()$
$\mathcal{S} := \emptyset$

Eavesdrop($m_L, m_R \in \Sigma.\text{M}$):
    $c := \Sigma.\text{Enc}(k_\Sigma, m_L)$
    $t :=$ challenger.GetTag($c$)
    $\mathcal{S} := \mathcal{S} \cup \{(c, t)\}$
    return $(c, t)$

Decrypt($(c, t) : c \in \Sigma.\text{C},\ t \in M.\mathcal{T}$)
    if $(c, t) \in \mathcal{S}$ return None
    if not challenger.CheckTag($c, t$) return None
    return $\Sigma.\text{Dec}(k_\Sigma, c)$
```

Now, thanks to the assumed unforgeability of M, we can replace the real unforgeability game for M with the random version. This is an indistinguishability hop demonstrating that $\mathbf{MAC}^{M}_{\text{real}} \circ \mathbf{R}^{\text{EtM}}_{1} \approx \mathbf{MAC}^{M}_{\text{rand}} \circ \mathbf{R}^{\text{EtM}}_{1}$

```
$\mathbf{MAC}^{M}_{\text{rand}}$

$k := M.\text{KeyGen}()$
$\mathcal{S} := \emptyset$

GetTag($m \in M.\mathcal{M}$):
    $t := M.\text{MAC}(k, m)$
    $\mathcal{S} := \mathcal{S} \cup \{(m, t)\}$
    return $t$

CheckTag($m \in M.\mathcal{M}, t \in M.\mathcal{T}$):
    return $(m, t) \overset{?}{\in} \mathcal{S}$
```

```
$\mathbf{R}^{\text{EtM}}_{1}$

$k_\Sigma := \Sigma.\text{KeyGen}()$
$\mathcal{S} := \emptyset$

Eavesdrop($m_L, m_R \in \Sigma.\text{M}$):
    $c := \Sigma.\text{Enc}(k_\Sigma, m_L)$
    $t :=$ challenger.GetTag($c$)
    $\mathcal{S} := \mathcal{S} \cup \{(c, t)\}$
    return $(c, t)$

Decrypt($(c, t) : c \in \Sigma.\text{C},\ t \in M.\mathcal{T}$)
    if $(c, t) \in \mathcal{S}$ return None
    if not challenger.CheckTag($c, t$) return None
    return $\Sigma.\text{Dec}(k_\Sigma, c)$
```

Next, we simply write an intermediate game $\mathbf{G}^{\text{EtM}}_{1}$ which inlines the previous reduction into one single game. Since both games have a set variable named $\mathcal{S}$, when inlining, we will label the challenger set as $\mathcal{S}_1$ and the reduction set as $\mathcal{S}_2$. This is an interchangeable step showing that $\mathbf{MAC}^{M}_{\text{rand}} \circ \mathbf{R}^{\text{EtM}}_{1} \equiv \mathbf{G}^{\text{EtM}}_{1}$.

```
$\mathbf{G}_1^{\mathrm{EtM}}$

$k_\Sigma := \Sigma.\mathrm{KeyGen}()$
$\mathcal{S}_1 := \emptyset$
$k_M := M.\mathrm{KeyGen}()$
$\mathcal{S}_2 := \emptyset$

Eavesdrop($m_L, m_R \in \Sigma.\mathrm{M}$):
    $c := \Sigma.\mathrm{Enc}(k_\Sigma, m_L)$
    $t := M.\mathrm{MAC}(k_M, c)$
    $\mathcal{S}_2 := \mathcal{S}_2 \cup \{(c, t)\}$
    $\mathcal{S}_1 := \mathcal{S}_1 \cup \{(c, t)\}$
    return $(c, t)$

Decrypt($(c, t) : c \in \Sigma.\mathrm{C},\ t \in M.\mathcal{T}$)
    if $(c, t) \in \mathcal{S}_1$ return None
    if $(c, t) \notin \mathcal{S}_2$ return None
    return $\Sigma.\mathrm{Dec}(k_\Sigma, c)$
```

We then can observe that throughout the entire game, $\mathcal{S}_1$ and $\mathcal{S}_2$ take on the same value. The game can be rewritten by unifying these two values back into a single set $\mathcal{S}$. In doing so, the decryption oracle will return None regardless of whether (c, t) is in $\mathcal{S}$ or not. Therefore, the entire decryption oracle can be also be rewritten to just return None. Both of these changes do not affect the output of any oracles, so this hop demonstrates interchangeability: $\mathbf{G}_1^{\mathrm{EtM}} \equiv \mathbf{G}_2^{\mathrm{EtM}}$

```
$\mathbf{G}_2^{\text{EtM}}$

$k_\Sigma := \Sigma.\text{KeyGen}()$
$k_M := M.\text{KeyGen}()$
$\mathcal{S} := \emptyset$

$\underline{\text{Eavesdrop}(m_L, m_R \in \Sigma.\text{M}):}$
    $c := \Sigma.\text{Enc}(k_\Sigma, m_L)$
    $t := M.\text{MAC}(k_M, c)$
    $\mathcal{S} := \mathcal{S} \cup \{(c, t)\}$
    $\textbf{return } (c, t)$

$\underline{\text{Decrypt}((c, t) : c \in \Sigma.\text{C},\ t \in M.\mathcal{T})}$
    $\textbf{return}$ None
```

Having removed the call to Σ.Dec, we can now rewrite the Σ.Enc call in terms of a reduction to the left CPA game for Σ. That is, we use an interchangeable hop, and show that $\mathbf{G}_2^{\text{EtM}} \equiv \mathbf{CPA}_L^\Sigma \circ \mathbf{R}_2^{\text{EtM}}$

```
$\mathbf{CPA}_L^\Sigma$

$k := \Sigma.\text{KeyGen}()$

$\underline{\text{Eavesdrop}(m_L, m_R \in \Sigma.\text{M}):}$
    $c := \Sigma.\text{Enc}(k, m_L)$
    $\textbf{return } c$
```

```
$\mathbf{R}_2^{\text{EtM}}$

$k_M := M.\text{KeyGen}()$
$\mathcal{S} := \emptyset$

$\underline{\text{Eavesdrop}(m_L, m_R \in \Sigma.\text{M}):}$
    $c :=$ challenger.Eavesdrop$(m_L,\ m_R)$
    $t := M.\text{MAC}(k_M, c)$
    $\mathcal{S} := \mathcal{S} \cup \{(c, t)\}$
    $\textbf{return } (c, t)$

$\underline{\text{Decrypt}((c, t) : c \in \Sigma.\text{C},\ t \in M.\mathcal{T})}$
    $\textbf{return}$ None
```

And from here, we can use the assumed CPA security of Σ to replace the left CPA game with the right CPA game, yielding $\mathbf{CPA}_L^\Sigma \circ \mathbf{R}_2^{\text{EtM}} \approx \mathbf{CPA}_R^\Sigma \circ \mathbf{R}_2^{\text{EtM}}$. Having replaced m_L

with m_R, we can apply the steps of the proof in reverse in order to hop to $\mathbf{CCA}_{\mathrm{R}}$. That is, we first write an intermediate game which inlines the reduction used in $\mathbf{CPA}_R^\Sigma \circ \mathbf{R}_2^{\mathrm{EtM}}$. Then, we split the variable $\mathcal{S}$ back into $\mathcal{S}_1$ and $\mathcal{S}_2$, and add the logic for decryption back in. This allows us to write a reduction to the random MAC game for M, which can then be swapped indistinguishably with the real MAC game for M by the assumed unforgeability of M. Finally, this reduction using a real MAC can be inlined to produce the exact game $\mathbf{CCA}_{\mathrm{R}}$. This sequence of hops yields that $\mathbf{CCA}_{\mathrm{L}} \approxeq \mathbf{CCA}_{\mathrm{R}}$ and so EtM is a CCA secure symmetric encryption scheme.

4.5.3 ProofFrog Encoding

This proof relies on some constructions for which the ProofFrog encoding has already been discussed: namely, symmetric encryption schemes and the CPA security definition. We present the ProofFrog files for the new definitions in the same order as presented earlier. First, we provide the model for CCA security in Figure 4.25. Next, we provide the definition of a MAC primitive in Figure 4.26. We also provide the pair of games for MAC unforgeability in Figure 4.27. The scheme file for the Encrypt-then-MAC construction is given in Figure 4.28. Finally, we list the reductions used in the proof in Figure 4.29, Figure 4.30, and Figure 4.31, and the proof file itself in Figure 4.32.

These examples also present parameterized sets which are a ProofFrog type that has not yet been mentioned. Sets are treated as collections of distinct values. ProofFrog supports notation for checking membership (`x in S`), checking whether one set is a subset of another (`S subsetof T`), creating unions of two sets (`S union T`), creating intersections of two sets (`S intersect T`), and getting the set difference of two sets (`S \ T`).

```
Game Left(SymEnc E) {
    E.Key k;
    Set<E.Ciphertext> S;
    Void Initialize() {
        k = E.KeyGen();
    }
    E.Ciphertext Eavesdrop(E.Message mL, E.Message mR) {
        E.Ciphertext c = E.Enc(k, mL);
        S = S union c;
        return c;
    }
    E.Message? Decrypt(E.Ciphertext c) {
        if (c in S) { return None; }
        return E.Dec(k, c);
    }
}
Game Right(SymEnc E) {
    E.Key k;
    Set<E.Ciphertext> S;
    Void Initialize() {
        k = E.KeyGen();
    }
    E.Ciphertext Eavesdrop(E.Message mL, E.Message mR) {
        E.Ciphertext c = E.Enc(k, mR);
        S = S union c;
        return c;
    }
    E.Message? Decrypt(E.Ciphertext c) {
        if (c in S) { return None; }
        return E.Dec(k, c);
    }
}
export as CCA;
```

Figure 4.25: ProofFrog syntax for the pair of games modelling CCA security for a symmetric encryption scheme.

```
Primitive MAC(Set MessageSpace, Set TagSpace, Set KeySpace) {
    Set Message = MessageSpace;
    Set Tag = TagSpace;
    Set Key = KeySpace;

    Key KeyGen();
    Tag MAC(Key k, Message m);
}
```

Figure 4.26: ProofFrog syntax for a MAC scheme.

```
Game Real(MAC M) {
    M.Key k;
    Void Initialize() {
        k = M.KeyGen();
    }
    M.Tag GetTag(M.Message m) {
        return M.MAC(k, m);
    }
    Bool CheckTag(M.Message m, M.Tag t) {
        return t == E.MAC(k, m);
    }
}
Game Random(MAC M) {
    E.Key k;
    Set<E.Message * E.Tag> S;
    Void Initialize() {
        k = M.KeyGen();
    }
    M.Tag GetTag(M.Message m) {
        M.Tag t = M.MAC(k, m);
        S = S union [m, t];
        return t;
    }
    Bool CheckTag(M.Message m, M.Tag t) {
        return [m, t] in S;
    }
}
export as Unforgeablility;
```

Figure 4.27: ProofFrog syntax for the pair of games modelling unforgeability for a MAC scheme.

```
Scheme EncryptThenMAC(SymEnc E, MAC M) extends SymEnc {
    requires E.Ciphertext subsetof M.Message;
    Set Key = E.Key * M.Key;
    Set Message = E.Message;
    Set Ciphertext = E.Ciphertext * M.Tag;
    Key KeyGen() {
        E.Key ke = E.KeyGen();
        M.Key me = M.KeyGen();
        return [ke, me];
    }
    Ciphertext Enc(Key k, Message m) {
        E.Ciphertext c = E.Enc(k[0], m);
        M.Tag t = M.MAC(k[1], c);
        return [c, t];
    }
    Message? Dec(Key k, Ciphertext c) {
        if (c[1] != M.MAC(k[1], c[0])) {
            return None;
        }
        return E.Dec(k[0], c[0]);
    }
}
```

Figure 4.28: ProofFrog syntax for the Encrypt-then-MAC construction.

```
Reduction R1(SymEnc E, MAC M, EncryptThenMAC EtM)
compose Unforgeability(M) against CCA(EtM).Adversary {
    E.Key ke;
    Set<E.Ciphertext * M.Tag> S;
    Void Initialize() {
        ke = E.KeyGen();
    }
    EtM.Ciphertext Eavesdrop(EtM.Message mL, EtM.Message mR) {
        E.Ciphertext c = E.Enc(ke, mL);
        M.Tag t = challenger.GetTag(c);
        S = S union [c, t];
        return [c, t];
    }
    EtM.Message? Decrypt(EtM.Ciphertext c) {
        if (c in S) {
            return None;
        }
        if (!challenger.CheckTag(c[0], c[1])) {
            return None;
        }
        return E.Dec(ke, c[0]);
    }
}
```

Figure 4.29: The first reduction used in the proof of Theorem 5.

```
Reduction R2(SymEnc E, MAC M, EncryptThenMAC EtM)
compose CPA(E) against CCA(EtM).Adversary {
    M.Key km;
    Set<E.Ciphertext * M.Tag> S;
    Void Initialize() {
        km = M.KeyGen();
    }
    EtM.Ciphertext Eavesdrop(EtM.Message mL, EtM.Message mR) {
        E.Ciphertext c = challenger.Eavesdrop(mL, mR);
        M.Tag t = M.MAC(km, c);
        S = S union [c, t];
        return [c, t];
    }
    EtM.Message? Decrypt(EtM.Ciphertext c) {
        if (c in S) {
            return None;
        }
        if (!(c in S)) {
            return None;
        }
    }
}
```

Figure 4.30: The second reduction used in the proof of Theorem 5.

```
Reduction R3(SymEnc E, MAC M, EncryptThenMAC EtM)
compose Unforgeability(M) against CCA(EtM).Adversary {
    E.Key ke;
    Set<E.Ciphertext * M.Tag> S;
    Void Initialize() {
        ke = E.KeyGen();
    }
    EtM.Ciphertext Eavesdrop(EtM.Message mL, EtM.Message mR) {
        E.Ciphertext c = E.Enc(ke, mR);
        M.Tag t = challenger.GetTag(c);
        S = S union [c, t];
        return [c, t];
    }
    EtM.Message? Decrypt(EtM.Ciphertext c) {
        if (c in S) {
            return None;
        }
        if (!challenger.CheckTag(c[0], c[1])) {
            return None;
        }
        return E.Dec(ke, c[0]);
    }
}
```

Figure 4.31: The third reduction used in the proof of Theorem 5.

```
proof:
let:
    Set SymEncKeySpace;
    Set MACKeySpace;
    Set MessageSpace;
    Set CiphertextSpace;
    Set TagSpace;
    SymEnc E = SymEnc(MessageSpace, CiphertextSpace, SymEncKeySpace);
    MAC M = MAC(CiphertextSpace, TagSpace, MACKeySpace);
    EncryptThenMAC EtM = EncryptThenMAC(E, M);
assume:
    CPA(E);
    Unforgeability(M);
theorem:
    CCA(EtM);
games:
    CCA(EtM).Left against CCA(EtM).Adversary;
    Unforgeability(M).Real compose R1(E, M, EtM) against CCA(EtM).Adversary;
    Unforgeability(M).Random compose R1(E, M, EtM) against CCA(EtM).Adversary;
    CPA(E).Left compose R2(E, M, EtM) against CCA(EtM).Adversary;
    CPA(E).Right compose R2(E, M, EtM) against CCA(EtM).Adversary;
    Unforgeability(M).Random compose R3(E, M, EtM) against CCA(EtM).Adversary;
    Unforgeability(M).Real compose R3(E, M, EtM) against CCA(EtM).Adversary;
    CCA(EtM).Right against CCA(EtM).Adversary;
```

Figure 4.32: Proof file for Theorem 5.

4.5.4 Simplify Not Operations

As previously mentioned, this proof is largely able to be verified utilizing our set of pre-existing transformations. The first hop, from `CCA(EtM)` to `Unforgeability(M).Real compose R1(E, M, EtM)`, utilizes instantiation and inlining to create an inlined game for the reduction. ProofFrog applies tuple expansion to the key tuple generated by `EtM.KeyGen` so that the two games both have independent `ke` and `km` fields. It also uses copy propagation to remove some extra variables created by the tuple expansion and inlining procedures, as well as using topological sorting to yield a canonical line ordering for the `Eavesdrop` method. After applying all of the simplifications, the games are near identical. The inlined game is shown below:

```
Game Inlined() {
  MACKeySpace challenger@k;
  SymEncKeySpace ke;
  Set<CiphertextSpace * TagSpace> S;
  Void Initialize() {
    challenger@k = M.KeyGen();
    ke = E.KeyGen();
  }
  CiphertextSpace * TagSpace Eavesdrop(MessageSpace mL, MessageSpace mR) {
    CiphertextSpace c = E.Enc(ke, mL);
    TagSpace t = M.MAC(challenger@k, c);
    S = S union [c, t];
    return [c, t];
  }
  MessageSpace? Decrypt(CiphertextSpace * TagSpace c) {
    if (c in S) {
      return None;
    }
    if (!(c[1] == M.MAC(challenger@k, c[0]))) {
      return None;
    }
    return E.Dec(ke, c[0]);
  }
}
```

The only difference (ignoring variable naming) between this and the AST created for `CCA(EtM).Left` is in the `Decrypt` oracle. The `CCA(EtM).Left` oracle has the condition `c[1] != M.MAC(k@1, c[0])`, whereas the inlined game uses the condition `!(c[1] == M.MAC(challenger@k, c[0]))`, as a byproduct of inlining the `challenger.CheckTag` call into `R1`. To canonicalize these two ASTs, we simply write a transformer that takes expressions of the form `!(A == B)` and rewrites them into `A != B`. After applying variable renaming, this one small transformation, in addition to all of the pre-existing transformations, allows us to verify the first hop of the proof. The second hop is verified as a use of an indistinguishability assumption. The third hop, from `Unforgeability(M).Random compose`

R1(E, M, EtM) to CPA(E).Left compose R2(E, M, EtM), does require some additional functionality.

4.5.5 Tuple Copies

First, consider the inlined game for Unforgeability(M).Random compose R1(E, M, EtM). Both games in this reduction define a set S, so the inlined game has both an S field and a challenger@S field. Just as in $\mathbf{G}_1^H$, these two sets take on the same value throughout their entire lifetime. ProofFrog is able to recognize this using the previously developed transformation for detecting duplicated fields, and is able to rewrite the game using just the S field. After applying this transformation to remove duplicated fields, and copy propagation, we have the following Decrypt oracle for Unforgeability(M).Random compose R2(E, M. EtM):

```
MessageSpace? Decrypt(CiphertextSpace * TagSpace c) {
  if (c in S) {
    return None;
  }
  if (!([c[0], c[1]] in S)) {
    return None;
  }
  return E.Dec(ke, c[0]);
}
```

This oracle AST differs from the Decrypt oracle in R2 in two crucial ways: first, that we use [c[0], c[1]] instead of just c, and second, that this method still has a call to E.Dec which must be removed. The first issue arises as a side effect of inlining: the CheckTag method expects a ciphertext and a tag, but in the random version of the MAC Unforgeability game, it just puts these two values back into a tuple again. Since the Decrypt oracle calls CheckTag with c[0] and c[1], inlining gives us [c[0], c[1]], which can clearly be simplified to just c. To bring the ASTs one step closer to canonicalization, we can write another simple transformer that just takes tuples of the form [a[0], a[1], ..., a[n]] and rewrite them as just a. Both ASTs now have matching checks to Decrypt which return None if c in S or if !(c in S). To achieve two identical Decrypt oracles just requires removing the final E.Dec call as it is unreachable code.

4.5.6 Unreachable Code

To determine that `E.Dec` is unreachable requires a slightly more complicated approach than the previous two transformers described in this section. The algorithm is described in Algorithm 2. Like some other previously described transformers, it uses Z3 to evaluate satisfiability of logical formulae.

The goal is to determine at which point in a block a return statement will definitely have been reached. Essentially, the formula f 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. Previously, we had mentioned that our ProofFrog AST to Z3 formula conversion will only produce a result if the operation is typed appropriately: the AST must use operations and types that have a one-to-one mapping in Z3. In this case, however, we know that each condition in an if-statement returns a boolean. So even though we cannot encode the individual parts of a condition like `c in S` in Z3 when using an abstract ciphertext type for `c`, we can still map the result of the operation `c in S` as a whole to a boolean in Z3. As such, each condition can be mapped to a Z3 formula, and if that condition being true would result in 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.

This transformation suffices to determine that the `E.Dec` call is unreachable code, and to verify interchangeability of the third hop: that `Unforgeability(M).Random compose R1(E, M, EtM)` is interchangeable with `CPA(E).Left compose R2(E, M, EtM)`.

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
            $f := f \vee (\neg l_0 \wedge \neg l_1 \wedge ... \wedge \neg l_n \wedge c)$, 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 $\neg f$ unsatisfiable then
        return all statements up to and including s
    end if
end for
return unchanged block
```

The fourth hop is an indistinguishability hop to replace `mL` with `mR`, and from there the proof proceeds in reverse. All further hops are verifiable with the strategies described so far. This example concludes our demonstration of the features of the proof engine. We present a final diagram showing all the transformations that ProofFrog uses in Figure 4.33.

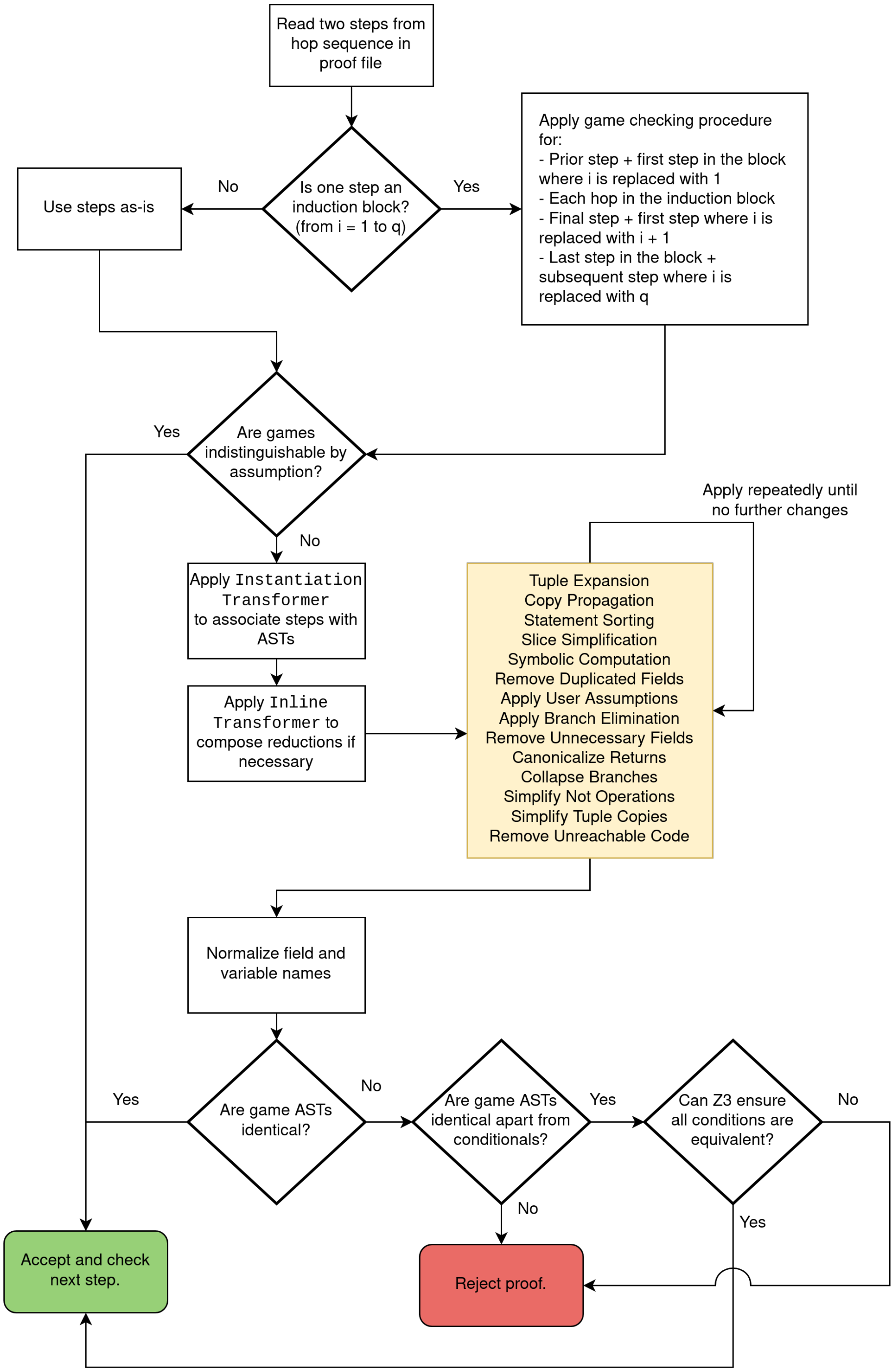
Read two steps from hop sequence in proof file
Is one step an induction block? (from i = 1 to q)
No
Use steps as-is
Yes
Apply game checking procedure for:
- Prior step + first step in the block where i is replaced with 1
- Each hop in the induction block
- Final step + first step where i is replaced with i + 1
- Last step in the block + subsequent step where i is replaced with q
Are games indistinguishable by assumption?
Yes
No
Apply Instantiation Transformer to associate steps with ASTs
Apply Inline Transformer to compose reductions if necessary
Apply repeatedly until no further changes
Tuple Expansion
Copy Propagation
Statement Sorting
Slice Simplification
Symbolic Computation
Remove Duplicated Fields
Apply User Assumptions
Apply Branch Elimination
Remove Unnecessary Fields
Canonicalize Returns
Collapse Branches
Simplify Not Operations
Simplify Tuple Copies
Remove Unreachable Code
Normalize field and variable names
Are game ASTs identical?
Yes
No
Are game ASTs identical apart from conditionals?
Yes
No
Can Z3 ensure all conditions are equivalent?
No
Yes
Accept and check next step.
Reject proof.

Figure 4.33: A flowchart of ProofFrog engine functionality necessary to prove Theorem 5.

Chapter 5

Conclusion

This thesis has presented ProofFrog: a new tool for verifying cryptographic game-hopping proofs. We have discussed in detail the tool's novel approach to proof verification via game AST canonicalization. We have demonstrated the efficacy of ProofFrog by encoding and proving a variety of theorems from *The Joy of Cryptography* [18], where each example demonstrated different features of the proof engine. ProofFrog also has the ability to prove a variety of other statements which were not presented for brevity, and because these proofs did not necessitate new functionality in the proof engine. Examples of such proofs include a proof that the one-time uniform ciphertexts property for symmetric encryption schemes implies the one-time secrecy property [18, Theorem 2.15], a proof demonstrating one-time secrecy of a construction using a secure PRG for symmetric encryption [18, Claim 5.4], a proof that a hybrid encryption construction is CPA secure [18, Claim 15.9], and others.

5.1 Future Work

One limitation of ProofFrog is that it requires trust from users. ProofFrog does have the ability to output each individual transformation that it performs, so that if a user does not trust ProofFrog claiming that a particular proof is valid, then the user can manually check each step for validity. But, manually checking each step for validity does to some extent defeat ProofFrog's purpose, which is to eliminate human review of proofs as much as possible. In an ideal world, any assertion that ProofFrog makes with regards to a proof's validity would be immediately accepted as fact. We did take care to ensure that ProofFrog only accepts valid proofs: numerous unit and functional tests were developed for the application to ensure correctness of AST transformations. Unfortunately, however, a formal calculus for the transformations ProofFrog uses and an accompanying proof of correctness was outside the scope of the project. As such, a user-base interested in formal verification may be wary of adopting an unproven engine. However, it is important to note that some other trusted proof assistants like CryptoVerif and EasyCrypt, while having formal calculi defined, also do not have formally checked implementations, so ProofFrog is not alone in requiring user trust. Nevertheless, a potential improvement to ProofFrog that would bolster confidence in the tool would be the ability to export the steps it takes to other proof assistant tools that work at a lower level of manipulating logical formulae, like EasyCrypt, for example. In doing so, the ProofFrog domain-specific language could also function as a higher level interface to proof assistants that are more widely trusted.

Another potential future avenue for work would be to expand the variety of examples that ProofFrog has been tested on. Aside from the example involving the double symmetric encryption scheme, each proof that ProofFrog has been able to verify was sampled from *The Joy of Cryptography* [18]. These proofs are noteworthy; the majority of these examples would likely be found in any other introductory cryptography textbook. However, there is of course a substantial gap between the complexity of proofs in textbooks versus those presented in research papers. It would be worthwhile to investigate whether the transformations developed for ProofFrog so far are sufficient for the majority of arguments made by an average cryptographer.

There are some constructions that could benefit the average cryptographer, but were not implemented in ProofFrog due to either complexity or time constraints. Pseudorandom functions (PRFs) are a common construction that could be useful in some proofs: security of a PRF allows a user to swap a real game that evaluates the PRF with a random game which uses a map that lazily samples and memoizes responses for later use. Arguments in cryptographic proofs often proceed by assuming that a PRF is secure, showing that inputs to the PRF do not repeat (or repeat with negligible probability), and from that one can conclude that each output is indistinguishable from just random sampling. Although ProofFrog's grammar does allow one to represent the definition of a PRF and model a security game using a map type, the engine currently struggles with making meaningful transformations to PRFs due to the long-lasting state persisting across oracle calls. This is a weakness of ProofFrog when compared to existing tools such as EasyCrypt and CryptoVerif. EasyCrypt supports the creation of PRFs via its map type provided in the standard library. A common example of a symmetric encryption scheme constructed from a PRF, and a proof that the scheme is CPA-secure, is found in Stoughton's EasyTeach repository [20]. The proof utilizes EasyCrypt's ability to reason probabilistically about failure events and the memory spaces that games would undertake, which ProofFrog currently does not support. CryptoVerif also supports PRFs by leveraging its novel calculus which preserves values of all variables across oracle calls. This preservation allows CryptoVerif to rewrite PRF lookups as searches across previous values of the program. If an input was previously provided to the PRF then it can fetch the corresponding response from that previous call, otherwise it can simply return a new randomly sampled value. The search across previously defined values can then sometimes be proven automatically false by the equational prover, which allows for simplification to purely random sampling. Analyzing the probability of failure events like in EasyCrypt strays from ProofFrog's design philosophy, therefore better support for PRFs in ProofFrog would likely require implementing stronger reasoning about the possible values variables can undertake. Because ProofFrog models PRFs via user-defined mutation to maps, determining the values of variables across oracle calls is more challenging than CryptoVerif's semantics where keeping track of the map's responses is done implicitly.

Another potential improvement could be to expand the number of built-in types to allow easier modelling of group elements and group operations, which are commonly used in the theory-heavy area of public-key cryptography. As of right now, a user would need to write a primitive to represent group elements, and then write manual reductions for each simple property of the group they would like to use. This is not too dissimilar from other cryptographic proof assistant tools. EasyCrypt provides a number of algebraic structures as part of its core library, including groups, rings, and fields. These are defined by specifying an abstract type that group elements belong to, specifying operations as functions across these types, and providing axioms and lemmas with which to manipulate these elements. EasyCrypt proofs involving these elements will require each axiom to be specifically applied on use, which is comparable to ProofFrog in its current state. However, the syntax for EasyCrypt is easier as applying an axiom just consists of writing that axiom's name as a tactic, whereas in ProofFrog, one would need to write a whole new reduction each time an axiom is applied. CryptoVerif similarly allows a user to define group elements by specifying abstract types and axioms of group operations via logical formulae that are then used by the equational prover. To bring ProofFrog up to par with these tools would likely require some additional language features that target simplification of group elements according to some pre-defined axioms.

Finally, there are a variety of tools that could be built on top of ProofFrog's domain-specific language that could benefit cryptographers. Despite type annotations existing inside of the language's grammar, a type checker has not yet been implemented, in part because a precise semantics for the ProofFrog language has not been specified. A type checker could act as a sanity check for cryptographers, ensuring that the reductions and games they write are actually well-formed. If a cryptographer has gone to the lengths of writing their proof inside of ProofFrog, it would also be a nicety if ProofFrog could automatically typeset the games by exporting to LaTeX, which would eliminate a tedious step of paper writing. It could also potentially export diagrams to layout the high-level steps of the proof; standardized proof diagrams could make papers more readable as one could easily grasp the high-level ideas of a proof at a glance. Finally, it could prove nice as a teaching tool to implement an interpreter that can actually execute the game ASTs as programs. This could allow one to write adversaries and demonstrate experimentally whether the adversary can distinguish which game they are composed with. Each of these tools could prove useful for cryptographers even if a user is not interested in the verification aspects of ProofFrog, or if their proof is too complex for ProofFrog to verify.

References

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

[2] David Aspinall. Proof General: A generic tool for proof development. In *Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems: Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000*, TACAS '00, page 38–42, Berlin, Heidelberg, 2000. Springer-Verlag.

[3] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. SoK: Computer-aided cryptography. In *2021 IEEE Symposium on Security and Privacy*, pages 777–795, San Francisco, CA, USA, May 24–27, 2021. IEEE Computer Society Press.

[4] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 71–90, Santa Barbara, CA, USA, August 14–18, 2011. Springer, Heidelberg, Germany.

[5] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Serge Vaudenay, editor, *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 409–426, St. Petersburg, Russia, May 28 – June 1, 2006. Springer, Heidelberg, Germany.

[6] Bruno Blanchet. A computationally sound mechanized prover for security protocols. In *2006 IEEE Symposium on Security and Privacy*, pages 140–154, Berkeley, CA, USA, May 21–24, 2006. IEEE Computer Society Press.

[7] Bruno Blanchet. Modeling and verifying security protocols with the applied pi calculus and ProVerif. *Foundations and Trends in Privacy and Security*, 1:1–135, 10 2016.

[8] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In Hugo Krawczyk, editor, *Advances in Cryptology – CRYPTO'98*, volume 1462 of *Lecture Notes in Computer Science*, pages 1–12, Santa Barbara, CA, USA, August 23–27, 1998. Springer, Heidelberg, Germany.

[9] Chris Brzuska, Antoine Delignat-Lavaud, Cédric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. State separation for code-based game-playing proofs. In Thomas Peyrin and Steven Galbraith, editors, *Advances in Cryptology – ASIACRYPT 2018, Part III*, volume 11274 of *Lecture Notes in Computer Science*, pages 222–249, Brisbane, Queensland, Australia, December 2–6, 2018. Springer, Heidelberg, Germany.

[10] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *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*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[11] Arthur B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5:558–562, 1962.

[12] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. CRC Press, third edition, 2020.

[13] Neal Koblitz and Alfred Menezes. Critical perspectives on provable security: Fifteen years of "another look" papers. *Advances in Mathematics of Communications*, 13(4):517–558, 2019.

[14] Matthew McKague and Douglas Stebila. pygamehop. https://github.com/dstebila/pygamehop.

[15] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The Tamarin prover for the symbolic analysis of security protocols. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 696–701, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[16] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. SymPy: symbolic computing in Python. *PeerJ Computer Science*, 3:e103, January 2017.

[17] Terence Parr, Sam Harwell, and Kathleen Fisher. Adaptive LL(*) parsing: the power of dynamic analysis. *SIGPLAN Not.*, 49(10):579–598, October 2014.

[18] Mike Rosulek. *The Joy of Cryptography*. https://joyofcryptography.com.

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

[20] Alley Stoughton. EasyTeach. https://github.com/alleystoughton/EasyTeach.

APPENDICES

Appendix A

ProofFrog Grammar

This appendix provides the grammar for each type of file that can be written in ProofFrog. Slight modifications have been made for increased legibility. In each grammar, parsing begins from the rule labelled `program`. In addition, each grammar depends on the `Shared` grammar, which as named, contains rules that are shared among multiple types of files.

A.1 Primitive Grammar

```
grammar Primitive;
import Shared;

program: PRIMITIVE ID '(' paramList? ')' '{' primitiveBody '}' EOF;
primitiveBody: ((initializedField|methodSignature) ';')+;
```

A.2 Scheme Grammar

```
grammar Scheme;
import Shared;

program: moduleImport* scheme EOF;
scheme: SCHEME ID '(' paramList? ')' EXTENDS ID '{' schemeBody '}';
schemeBody: (REQUIRES expression ';')* (field ';' | method)+;
```

```
REQUIRES: 'requires';
SCHEME: 'Scheme';
EXTENDS: 'extends';
```

A.3 Game Grammar

```
grammar Game;
import Shared;

program: moduleImport* game game gameExport EOF;
gameExport: EXPORT AS ID ';';

EXPORT: 'export';
```

A.4 Proof Grammar

```
grammar Proof;
import Shared;

program: moduleImport* proofHelpers proof EOF;
proofHelpers: (reduction | game)*;
reduction: REDUCTION ID '(' paramList? ')'
    COMPOSE parameterizedGame AGAINST gameAdversary '{' gameBody '}';
proof: PROOF ':' (LET ':' lets)? (ASSUME ':' assumptions)?
    THEOREM ':' theorem  GAMES ':' gameList;
lets: (field ';')*;
assumptions: (parameterizedGame ';')* (CALLS ('<='|'<') expression ';')?;
theorem: parameterizedGame ';';
gameList: gameStep ';' (gameStep ';'|induction|stepAssumption)*;
gameStep: concreteGame COMPOSE parameterizedGame AGAINST gameAdversary
        | (concreteGame|parameterizedGame) AGAINST gameAdversary
        ;
induction: INDUCTION '(' ID FROM integerExpression TO integerExpression ')'
    '{' gameList '}';
```

```
stepAssumption: ASSUME expression ';';
gameField: (concreteGame | parameterizedGame) '.' ID;
concreteGame: parameterizedGame '.' ID;
gameAdversary: parameterizedGame '.' ADVERSARY;

REDUCTION: 'Reduction';
AGAINST: 'against';
ADVERSARY: 'Adversary';
COMPOSE: 'compose';
PROOF: 'proof';
ASSUME: 'assume';
THEOREM: 'theorem';
GAMES: 'games';
LET: 'let';
CALLS: 'calls';
INDUCTION: 'induction';
FROM: 'from';
```

A.5 Shared Grammar

```
grammar Shared;

game: GAME ID '(' paramList? ')' '{' gameBody '}';
gameBody: (field ';')* method+
        | (field ';')* method* gamePhase+;
gamePhase: PHASE '{' (method)+ ORACLES ':' '[' id (',' id)* ']' ';' '}';
field: variable ('=' expression)?;
initializedField: variable '=' expression;
method: methodSignature block;
block: '{' statement* '}';
statement: type id ';'
        | type lvalue '=' expression ';'
        | type lvalue '<-' expression ';'
        | lvalue '=' expression ';'
        | lvalue '<-' expression ';'
        | expression '(' argList? ')' ';'
        | RETURN expression ';'
        | IF '(' expression ')' block (ELSE IF '(' expression ')' block )*
            (ELSE block )?
        | FOR '(' INTTYPE id '=' expression TO expression ')' block
        | FOR '(' type id IN expression ')' block
        ;
lvalue: (id | parameterizedGame) ('.' id | '[' integerExpression ']')*;
methodSignature: type id '(' paramList? ')';
paramList: variable (',' variable)*;
expression:
        expression '(' argList? ')' #fnCallExp
        | expression '[' integerExpression ':' integerExpression ']' #sliceExp
        | '!' expression #notExp
        | '|' expression '|' #sizeExp

        | expression '*' expression #multiplyExp
        | expression '/' expression #divideExp
        | expression '+' expression #addExp
        | expression '-' expression #minusExp
        | expression '==' expression #equalsExp
        | expression '!=' expression #notequalsExp
        | expression '>' expression #gtExp
        | expression '<' expression #ltExp
        | expression '>=' expression #geqExp
        | expression '<=' expression #leqExp
        | expression IN expression #inExp
        | expression SUBSETOF expression #subsetsExp

        | expression '&&' expression #andExp
        | expression '||' expression #orExp
        | expression UNION expression #unionExp
         | expression INTERSECT expression #intersectExp
        | expression '\\' expression #setMinusExp

        | lvalue # lvalueExp
        | '[' (expression (',' expression)*)? ']' #createTupleExp
        | '{' (expression (',' expression)*)? '}' #createSetExp
        | type #typeExp
        | BINARYNUM #binaryNumExp
        | INT #intExp
        | bool #boolExp
        | NONE #noneExp
        | '(' expression ')' #parenExp
        ;
argList: expression (',' expression)*;
variable: type id;
parameterizedGame: ID '(' argList? ')';
type: type '?' #optionalType
        | set #setType
        | BOOL #boolType
        | VOID #voidType
        | MAP '<' type ',' type '>' #mapType
        | ARRAY '<' type ',' integerExpression '>' #arrayType
        | INTTYPE #intType
        | type ('*' type)+ #productType
        | bitstring #bitStringType
        | lvalue # lvalueType
        ;
integerExpression
        : integerExpression '*' integerExpression
        | integerExpression '/' integerExpression
        | integerExpression '+' integerExpression
        | integerExpression '-' integerExpression
        | lvalue
        | INT
        | BINARYNUM
        ;
bitstring: BITSTRING '<' integerExpression '>' | BITSTRING;
set: SET '<' type '>' | SET;
bool: TRUE | FALSE;
moduleImport: IMPORT FILESTRING (AS ID)? ';';
id: ID | IN;

SET: 'Set';
BOOL: 'Bool';
VOID: 'Void';
```

```
INTTYPE: 'Int';
MAP: 'Map';
RETURN: 'return';
IMPORT: 'import';
BITSTRING: 'BitString';
ARRAY: 'Array';
PRIMITIVE: 'Primitive';
SUBSETOF: 'subsetof';
IF: 'if';
FOR: 'for';
TO: 'to';
IN: 'in';
UNION: 'union';
INTERSECT: 'intersect';
GAME: 'Game';
AS: 'as';
PHASE: 'Phase';
ORACLES: 'oracles';
ELSE: 'else';
NONE: 'None';
TRUE: 'true';
FALSE: 'false';
BINARYNUM: '0b'[01]+ ;
INT: [0-9]+ ;
ID: [a-zA-Z_$][a-zA-Z_0-9$]* ;
WS: [ \t\r\n]+ -> skip ;
LINE_COMMENT : '//' .*? '\r'? '\n' -> skip ;
FILESTRING: '\''[0-9a-zA-Z_$/.=>< ]+'\'' ;
```