Technische Universiteit Eindhoven University of Technology

Department of Mathematics and Computer Science
Coding Theory and Cryptology Group

Formal Verification of Saber

Master's Thesis

M.C.F.H.P. Meijers
Supervisor: dr. Andreas T. Hülsing

Final version

Thursday 23rd September, 2021
Eindhoven

Abstract

In this thesis, we consider the formal verification of (the specification of) the public-key encryption scheme provided in Saber, one of the selected few post-quantum cipher suites currently eligible for potential standardization. Specifically, we carry out a formal verification process concerning the desired security and correctness properties of Saber's public-key encryption scheme in the EasyCrypt tool. The purpose of this undertaking is to attain more assurance regarding the Saber cipher suite and its properties, assisting the cryptographic community in making a well-informed decision on the standardization of Saber.

Prior to the actual formal verification endeavor in EasyCrypt, we perform an extensive manual analysis of Saber's public-key encryption scheme, constructing hand-written proofs for the scheme's security and correctness properties. We purposely structure these proof in a manner that facilitates their formal verification. Namely, regarding the security proof, we adopt the code-based, game-playing approach to the provable security paradigm; this is the principal proof method supported by EasyCrypt. Furthermore, concerning the correctness proof, we devise an alternative specification of Saber's public-key encryption scheme that is equivalent to its original counterpart; indeed, this alternative specification admits a considerably less strenuous formal verification of the desired correctness property.

Leveraging the deliberate structure of the hand-written proofs, the formal verification effort for the desired security and correctness properties closely resembles these proofs. For both of the considered properties, the results of this formal verification effort are affirmative; that is, these results indicate that Saber's public-key encryption scheme indeed satisfies the desired security and correctness properties.

Preface

This document comprises the master's thesis "Formal Verification of Saber", which, fundamentally, is concerned with corroborating Saber's possession of the desired properties. The rationale for effectuating this thesis arises from the urgency of replacing our contemporary public-key cryptography with post-quantum alternatives; namely, Saber is a post-quantum cipher suite presently considered for potential standardization as a partial substitute for the current public-key cryptography. However, due to the limited amount of time and resources, the research in this thesis does not comprehensively cover the entirety of Saber; instead, it solely considers the most imperative and fundamental components of the cipher suite. This thesis constitutes part of the graduation requirements for a master's degree in Information Security Technology, a specialized master's track within the Computer Science and Engineering master's program, jointly provided by the Eindhoven University of Technology and the Radboud University.

At this point, I would like to thank several people for the assistance and opportunities they have provided me throughout this project; without them, this thesis would have been significantly more arduous and uneventful. First and foremost, I want to express my gratitude toward my supervisor, Andreas Hülsing, for numerous things; in particular, among others, these include granting me the opportunity to carry out this thesis, answering all of my questions on an extensive range of varying subjects, and generally guiding and supporting me throughout the entire process. Second, I would like to thank Pierre-Yves Strub for answering each of my questions on EasyCrypt and directly contributing to my code. Third, I want to show my appreciation to Jan-Pieter D'Anvers for helping me comprehend Saber by answering all of my queries and giving me access to an early version (of a chapter) of his dissertation. Fourth, I would like to show my gratitude to Peter Schwabe for first exposing me to the field of computer-aided cryptography and the Saber cipher suite by offering me a research internship on the high-speed and high-assurance implementation of Saber in Jasmin; moreover, I want to thank him for additionally being part of this thesis as a member of the assessment committee. Penultimately, I would like to show my appreciation to the teams working on Saber's and Kyber's encompassing formal verification projects for allowing me to give a talk introducing these projects and their underlying process at NIST's third PQC standardization conference. In the context of this talk, I especially want to thank Manuel Barbosa, Andreas Hülsing, and Peter Schwabe for giving me feedback on the slides, abstract, and presentation. Lastly, I would like to thank Erik de Vink for his willingness to join my assessment committee.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Cryptography is the scientific field concerned with guaranteeing the security of information, both in communication and storage. In this context, “security” may refer to any desirable properties that one wants to assure and preserve in the presence of an unauthorized, potentially malicious third party. Originally, confidentiality was the only such property that cryptography regarded; nevertheless, over time, the set of considered properties has extended significantly. Examples of some relatively prominent properties that contemporary cryptography concerns in addition to confidentiality are authenticity, integrity, and non-repudiation [1].

At present, in an attempt to satisfactorily achieve the above-mentioned security in numerous different contexts, the field of cryptography has advanced to the point where a plethora of cryptographic constructions is readily available for deployment. Fundamentally, all of these constructions are comprised of (a combination of) cryptographic primitives. A prevalent category of such primitives is the category of *public-key primitives.* Intrinsic to the primitives in this category is that they require two mathematically related artifacts to function correctly: a *public key* and a *private key.* As their names suggest, the former key may be openly distributed to every entity in the considered communication medium, while the latter key must remain exclusive to the owner of the key pair. Utilizing the mathematical relation between the keys and the fact that nobody but the rightful owner has access to the private key, the primitive intends to provide a certain functionality satisfying a set of security properties. Indeed, the security of such a primitive therefore wholly relies on the fact that knowledge of the public key (or any other public information that the primitive produces) does not provide any information about the private key nor enables the violation of the primitive’s security properties in some other manner. In turn, this is entirely contingent on the hardness of the mathematical problem(s) underlying the primitive’s operations and the relationship between the keys. Currently, nearly all of the public-key primitives are based on the integer factorization or discrete logarithm problem [2]. Albeit these problems are hard to solve for *classical computers*, i.e., computers operating on classical bits, they are trivial to solve for *quantum computers*, i.e., computers operating on quantum bits [3]. Consequently, an operational quantum computer would be able to compromise the security of all such public-key primitives and, as a result, the currently employed constructions that utilize these primitives [4]. Since the advent of operational quantum computers is an envisioning that is likely to become a reality, a timely replacement of the contemporary cryptographic constructions reliant on public-key primitives with quantum-resistant alternatives is imperative [5].

Generally, confidence in the fact that cryptographic constructions possess the desired properties is established through the extensive scrutinization of these constructions and their proofs, predominantly performed by the cryptographic community. Nevertheless, despite this extensive scrutinization, faulty constructions and incorrect proofs frequently go unnoticed for extended periods of time, as exemplified by a multitude of instances throughout history [6]. In part, these issues instigated the inception of the scientific field of *computer-aided cryptography.* This field seeks to develop approaches to the construction and verification of cryptography that employ computers to formally guarantee the veracity of claims related to the desired properties of cryptographic constructions [7]. The utilization of computers in this manner assists in reducing the complexity of the construction and verification effort while simultaneously providing a consistently high level of rigorousness.

Combining the subjects introduced above, this thesis concerns the computer-aided (or formal) verification of the security and correctness of several cryptographic constructions from a post-quantum cryptographic suite that utilizes quantum-resistant public-key primitives. Expanding on this, the remainder of this chapter is organized as follows. First, Section 1.1 and Section 1.2 further introduce the fields of post-quantum cryptography and computer-aided cryptography; specifically, these sections extend the above introductions and elaborate on the more specific parts of these fields relevant to this thesis. Second, Section 1.3 discusses some of the earlier and, at the time of writing, ongoing work related to the work presented here. Third, Section 1.4 explains the purpose of this thesis and argues for the significance of its contribution to the field of cryptography. Lastly, Section 1.5 provides an overview of the rest of this thesis.

1.1 Post-Quantum Cryptograhpy

Over the past few decades, a considerable amount of research has been conducted on the feasibility, applications, and consequences of quantum computing. This research has led to plentiful significant discoveries, many of which are related to the protection of digital information. In particular, one of these discoveries asserts that, as soon as sufficiently powerful quantum computers become operational, a substantial part of our contemporary cryptography is trivial to break; this assertion primarily concerns the current public-key cryptography [8, 9]. As a result, in the presence of such quantum computers, it would be impossible to ensure the security of information through this public-key cryptography. Moreover, information communicated or stored at present, even if secure by current standards, is at risk of having its security compromised in the future; indeed, if such information is preserved until an adequate quantum computer becomes available, the security of this information can trivially be compromised [10]. Although it is not entirely clear when the first sufficiently powerful quantum computer will be operational, the tremendous progress made hitherto, the currently remaining challenges, and the exceptional amount of interest in this topic suggest that this could well transpire within the next several decades [5, 11]. Considering the standardization and ubiquitous deployment of post-quantum (i.e., quantum-resistant) alternatives to the contemporary public-key cryptography will bring about numerous novel challenges that require a significant amount of time and effort to resolve, it is of utmost importance to commence this process promptly [12].

1.1.1 Saber

At the time of writing, the National Institute of Standards and Technology (NIST) is hosting a competition with the purpose of standardizing post-quantum alternatives to the current public-key cryptography [13]. Recently, this competition has advanced to its final round, leaving only a selected few of the best candidates. One of these final candidates is *Saber*, a suite of post-quantum cryptographic constructions for public-key encryption and key-establishment. Specifically, Saber consists of a Key Exchange (KE) scheme, a Public-Key Encryption (PKE) scheme, and a Key Encapsulation Mechanism (KEM) [14]. These schemes are closely related due to the fact that the KEM encompasses the PKE scheme, and the PKE scheme is based on the KE scheme. Despite this close relation, each of these schemes attempts to fulfill a unique security property: the KE scheme strives to generate keys that are indistinguishable from random keys, a property termed

"IND-RND security"; the PKE scheme intends to provide ciphertext Indistinguishability under Chosen-Plaintext Attack (IND-CPA), a property commonly referred to as "IND-CPA security"; and the KEM aims to yield ciphertext Indistinguishability under Adaptive Chosen-Ciphertext Attack (IND-CCA2), a property generally denominated "IND-CCA2 security"[1].

As aforementioned, the security of cryptographic constructions utilizing public-key primitives is contingent on the hardness of the primitive's underlying mathematical problem; the schemes from the Saber cipher suite are no exception. Furthermore, as Saber's schemes are supposed to be quantum-resistant, the problem underlying each scheme must not be efficiently solvable by both classical and quantum computers. Since these schemes are built on top of each other, the underlying mathematical problem is the same for all of them; this problem is dubbed the Module-Learning With Rounding (MLWR) problem, a variant of the Learning With Rounding (LWR) problem [15]. Indeed, even considering the possibility of quantum computing, no efficient solving method exists for these problems [16]. Combined with the other design decisions, this choice of mathematical problem aspires to maximize the schemes' simplicity, efficiency, and flexibility [14].

At the end of its competition, NIST will presumably exclusively standardize the KEMs of the victors, establishing Saber's KEM as the predominant scheme of the Saber cipher suite. This thesis directs its attention at the unique core component of Saber's KEM, i.e., Saber's PKE scheme, aspiring to analyze and formally verify the proposed security and correctness properties of this PKE scheme. Indeed, due to the composition of Saber's KEM, its properties are entirely contingent on the security and correctness properties of Saber's PKE scheme. As such, although Saber's PKE scheme will probably *not* be standardized as an independent scheme, the formal verification of its properties still has considerable merit.

1.2 Computer-Aided Cryptography

Historically, cryptographic constructions have been demonstrated to possess their desired properties by means of hand-written proofs. However, the innovation and development in the field of cryptography have led to a significant increase in the complexity of these constructions and their proofs; as such, hand-written proofs have become substantially more challenging to carry out correctly. Multiple instances of proofs exist that, although extensively scrutinized and universally considered correct, turned out to be faulty. Furthermore, in some of these cases, the corresponding cryptographic construction was additionally found to be insecure [6]. These instances clearly exemplify the intricacy of properly contriving and verifying cryptographic constructions and their proofs.

In addition to the above concern, even if a cryptographic construction and its proof are entirely correct, implementation flaws may invalidate any of the construction's properties and guarantees. Alternatively stated, despite the soundness of a construction's specification and associated proof, none of the desired properties might actually be achieved by a faulty implementation. Consequently, such an implementation may, for example, not provide the level of security that the corresponding construction intends to provide, allowing a malicious entity to compromise the security of any information processed by this implementation. As with the previous concern, ample examples exist of this phenomenon, signifying the importance of proper cryptographic implementations [17].

As alluded to before, the above-mentioned issues partially induced the establishment of the scientific field of computer-aided cryptography. This field endeavors to devise methods for the construction and verification of cryptography that employ computers to formally guarantee the veracity of claims related to the desired properties of both specifications and implementations [7]. The purpose of these computer-assisted methods is to reduce the complexity of the manual labor

[1]If any of these concepts seems unfamiliar, refer to Chapter 2 for an explanation and further insight.

required in the construction and verification process while consistently enforcing exceptional rigorousness. In turn, this increases confidence in the specifications and implementations that are devised and analyzed in this manner.

1.2.1 EasyCrypt

Over the years, the research conducted in the field of computer-aided cryptography has produced copious tools and frameworks that utilize computers to facilitate the construction and verification of cryptography in a multitude of different ways and contexts [7]. Considering the context in which Saber's PKE scheme and the corresponding proofs manifests themselves, the tool of choice for this thesis is EasyCrypt.

EasyCrypt is a tool predominantly aimed at the formal verification of the security properties of cryptographic constructions [18]. To this end, the tool adopts the code-based approach, modeling common security-related concepts, such as security properties and hardness assumptions, as probabilistic programs [19]. Moreover, the tool's higher-order ambient logic, standard library, and other built-in mechanisms allow for, among others, extensive mathematical reasoning, the realization of different types of proofs, and the modular composition of cryptographic constructions. Due to this voluminous set of features, EasyCrypt's capabilities exceed the mere formal verification of security properties; particularly, the tool is additionally capable of verifying other customary properties of cryptographic constructions, e.g., correctness properties. As will become apparent in the subsequent chapters, the above-mentioned approach and expressiveness make EasyCrypt quite apt for the formal verification of Saber's PKE scheme.

1.3 Related Work

Although this thesis comprises the first publicly known formal verification endeavor regarding (the specification of) Saber's PKE scheme, closely related work has already been carried out or is currently in progress. Presently, we briefly discuss the most notable and relevant of this work.

Previously in my studies, i.e., for my research internship, I implemented Saber's PKE scheme and KEM employing Jasmin[2] [20]. *Jasmin* is a framework designed for the implementation of high-assurance and high-speed cryptography [21]; this framework consists of three primary components: a programming language, a certified compiler, and a set of verification tools. Noteworthy, this latter component comprises multiple tools that, given a program written in Jasmin's programming language, produce specifically constructed EasyCrypt code. Subsequently, this code can be analyzed in EasyCrypt to formally verify a particular property that is typically required from cryptographic implementations; specifically, depending on the tool utilized to generate the code, this concerns the constant-time and functional correctness properties [22]. In addition to these tools, Jasmin contains a tool that, instead of generating EasyCrypt code, fully automatically verifies the memory safety of a given Jasmin implementation; naturally, this is also a customary property expected from cryptographic implementations [22]. As such, EasyCrypt and Jasmin can be employed in conjunction to effectuate a general formal verification process, enabling the formal verification of both the specifications and implementations of cryptographic constructions. In fact, the work performed in this thesis and the aforementioned research internship is part of an encompassing project that, following this general formal verification process based on EasyCrypt and Jasmin, aspires to formally verify the specifications of Saber's PKE scheme and KEM, as well as construct formally verified, optimized implementations of these schemes.

Analogous to the above-mentioned encompassing project for Saber, a formal verification project is currently ongoing for Kyber, a post-quantum cipher suite consisting of an IND-CPA secure

[2]The corresponding GitHub repository can be found at: `https://github.com/MM45/SABER-Jasmin`.

PKE scheme and an IND-CCA2 secure KEM [23]. Similarly to Saber, Kyber is a finalist of NIST's post-quantum cryptography competition in the category for public-key encryption and key-establishment [13]. The formal verification project for Kyber follows the same process as the project for Saber; that is, utilizing EasyCrypt and Jasmin, the formal verification project for Kyber attempts to formally verify the specifications of Kyber's PKE scheme and KEM, as well as devise formally verified, optimized implementations of these schemes.

With the purpose of introducing the above-mentioned formal verification process and projects to the cryptographic community, I gave a talk at NIST's third post-quantum cryptography standardization conference [24]. Refer to this talk for more information on these subjects.

1.4 Purpose and Contribution

As indicated in the preceding sections, this thesis considers the formal verification of the security and correctness properties of Saber's PKE scheme. The purpose of this endeavor is to establish a higher level of confidence in the (in)security and (in)correctness of this scheme and, by extension, Saber's KEM. Namely, Saber's KEM is constructed from Saber's PKE scheme through a generic transformation, i.e., a variant of the so-called Fujisaki-Okamoto (FO) transformation. A crucial aspect of this transformation is that the properties of the resulting KEM are contingent on the properties of the initial PKE scheme. In particular, considering Saber's case, for Saber's KEM to be IND-CCA2 secure, Saber's PKE scheme must be IND-CPA secure and sufficiently correct; furthermore, the correctness of Saber's KEM is identical to that of Saber's PKE scheme [14, 25]. Suggesting its validity, this transformation has been formally verified by independent previous work [26]. As such, by formally verifying the security and correctness properties of Saber's PKE scheme, the confidence in the desired properties of both Saber's PKE scheme and KEM increases; conversely, the confidence in the desired properties of both schemes decreases if the formal verification effort suggests that Saber's PKE scheme does not satisfy its conjectured security and correctness properties.

A caveat of leveraging the result of the above-mentioned formal verification effort concerning the FO transformation to argue for the correctness and security of Saber's KEM is that this effort has not been carried out in EasyCrypt; instead, it has been performed in the *qRHL* tool [26]. Consequently, albeit the result of this effort suggests the transformation's correctness, it is not immediately apparent that this result can directly be integrated with the results of this thesis; this is predominantly due to the nontrivial syntactical and semantical differences between qRHL and EasyCrypt. As such, to reduce the risk of reaching unwarranted conclusions, this integration should be substantiated by a sound justification that corroborates the compatibility of the results. Indeed, given the complexity of the disparities between the considered tools, such a justification is presumably quite intricate and difficult to (manually) verify. Naturally, this intricacy enhances the potential of constructing an erroneous justification; if practicable, formally verifying the justification might partially alleviate this issue. Completely circumventing the predicaments pertinent to the construction of such a sound justification, the project that encompasses this thesis's work envisions to formally verify the desired properties of Saber's KEM in EasyCrypt.

Thus far, no other formal verification endeavors regarding Saber's schemes have been carried out (or, at least, no such endeavors are publicly known). Indeed, at the time of writing, Saber's schemes have solely been proved secure and correct through extensively scrutinized hand-written proofs. However, as explicated above, this entirely manual process is relatively error-prone and unrigorous compared to the computer-assisted scrutinization approach of formal verification. Hence, the work carried out in this thesis serves its purpose; that is, the performed work establishes a higher level of confidence in the (in)security and (in)correctness of Saber's PKE scheme and, by extension, Saber's KEM. Since this assists the cryptographic community in making a well-informed decision on whether to standardize Saber as a post-quantum cipher suite, this thesis constitutes a significant contribution to the field of cryptography.

1.5 Overview

In total, this thesis comprises five chapters of which this introductory chapter is the initial one; hence, the remainder of the thesis consists of four additional chapters. The following constitutes an overview of these chapters, concisely introducing their order, titles, and content.

- **Chapter 2 - Background Knowledge**
  Discusses the background knowledge required to facilely understand the discussion in the succeeding chapters. In particular, this chapter examines several subjects from the fields of mathematics, cryptography, and computer-aided cryptography.

- **Chapter 3 - Saber**
  Elaborates on Saber and, particularly, its PKE scheme. More precisely, this chapter introduces the context in which Saber manifests itself, explains the specification of Saber's PKE scheme, and performs (manual) analyses concerning the security and correctness properties of Saber's PKE scheme.

- **Chapter 4 - Formal Verification**
  Covers the formal verification effort carried out for Saber's PKE scheme. Specifically, this chapter considers the formalization of the relevant concepts and artifacts and, utilizing these formalizations, expands on the formal verification of Saber's PKE scheme with respect to its security and correctness properties. Furthermore, this chapter gives a demonstration on the (lemma-)proving process in EasyCrypt by explicating the concrete proof of a single lemma employed in the formal verification effort concerning Saber's PKE scheme.

- **Chapter 5 - Conclusion**
  Draws conclusions from the material presented in this thesis and discusses potential future work.

Chapter 2

Background Knowledge

Following the introductory chapter, this chapter provides the background knowledge necessary to comprehend the discussions presented throughout this thesis facilely; in addition, this chapter specifies the notational guidelines and conventions. More precisely, the provided background knowledge comprises several subjects from the fields of mathematics, cryptography, and computer-aided cryptography. Although the coverage of these subjects should provide sufficient information to follow the remainder of this thesis comfortably, it does not encompass all material related to Saber and its formal verification. In particular, certain rudimentary concepts within the relevant branches of mathematics and computer science are merely concisely summarized, primarily serving as a brief reminder, or not elaborated on at all.

The remainder of this chapter is structured as follows. First, Section 2.1 explains the mathematics that underlies Saber and its analysis; this explanation covers topics from the branches of (abstract) algebra, number theory, and probability theory. Second, Section 2.2 introduces the relevant notions and concepts from the field of cryptography; among others, this section discusses the subjects of public-key cryptography, provable security, and lattice-based cryptography. Third, Section 2.3 elaborates on material related to computer-aided cryptography, primarily fixating on the Easy-Crypt tool. Lastly, Section 2.4 describes the notational guidelines and conventions considered throughout.

2.1 Mathematics

The material discussed in this section and more elaborate explications thereof can be found in most elementary books on the considered subjects. Examples of such books are the following: regarding abstract algebra, [27] and [28]; concerning number theory, [29] and [30]; and for probability theory, [31].

2.1.1 Abstract Algebra and Number Theory

Several topics from the branches of abstract algebra and number theory are fundamental to the work in this thesis; most notably, this concerns the topics of modular arithmetic and algebraic structures based on this arithmetic. In the ensuing, we briefly recapitulate the relevant rudimentary material from these topics and, afterward, elaborate on the manifestation of these topics in this thesis.

Euclidean Division and the Modulo Operation

With respect to integers, *Euclidean division* is the division of an integer, i.e., the dividend, by another integer, i.e., the divisor, producing a *quotient* and a *remainder*. Here, both the quotient and the remainder are integers; moreover, the remainder lies between 0 (including) and the absolute value of the divisor (excluding). An important property of Euclidean division is that, assuming the divisor does not equal 0, the quotient and remainder always exist and are unique. More formally, given two integers x and y, where $y \neq 0$, there exists unique integers q (quotient) and r (remainder) such that $0 \leq r < |y|$ and the following equation holds.

$$x = q \cdot y + r$$

Intimately related to Euclidean division, the *modulo* operation, denoted by mod, is solely concerned with computing the above-discussed remainder; i.e., considering the same context as above, the ensuing equation is veracious.

$$r = x \bmod y$$

Modular Arithmetic

Modular arithmetic refers to a system of arithmetic (for integers) that is defined with respect to a particular strictly positive integer, the *modulus*. In such a system, two integers are said to be *congruent* if their difference is an integer multiple of the considered modulus. That is, given integers x, y, k, and n, where $1 < n$, then x and y are congruent with respect to modulus n if $x - y = k \cdot n$; equivalently, x and y are congruent with respect to modulus n if $x \bmod n = y \bmod n$. Grouping the integers based on this congruence relation, we obtain a finite number of distinct integer collections; these collections are referred to as congruence classes and, together, they comprise all integers.

The fundamental operations in a modular arithmetic system are identical to those for the integers; nevertheless, the computations in a modular arithmetic system are carried out on the above-mentioned congruence classes instead of on integers. Practically, performing a computation in a modular arithmetic system is straightforwardly accomplished by, for each considered congruence class, choosing an arbitrary integer from this class and, subsequently, carrying out the computation with these integers. Irrespective of the representative integer chosen for each congruence class, the outcome of this computation invariably belongs to the same congruence class.

Sets, Groups, and Rings

In mathematics, a *set* can refer to any collection of distinct mathematical objects. Building on this concept, a *group* is a set with a binary operator, commonly referred to as “addition”, satisfying the following axioms. In describing these axioms, we denote the group’s set and binary operator by S and $+$, respectively.

- *Closure*
  For all $x, y \in S$, we have $x + y \in S$.

- *Associativity*
  For all $x, y, z \in S$, we have $(x + y) + z = x + (y + z)$.

- *Identity Element*
  There exists a unique element $e \in S$ such that, for all $x \in S$, we have $e + x = x + e = x$.

- *Inverse Element*
  For all $x \in S$, there exists a $y \in S$ such that $x + y = e$, where e is the identity element defined in the previous axiom.

In case the group's addition operator moreover satisfies the commutativity axiom, then the group is called a *commutative group* or an *abelian group*; this axiom is defined below.

- *Commutativity*
  For all $x, y \in S$, we have $x + y = y + x$.

Harnessing the concept of an abelian group, a *ring* is an abelian group with a second binary operator, customarily denominated "multiplication", defined on the corresponding set. Naturally, besides conforming to the abelian group axioms regarding its addition operator, a ring additionally satisfies several axioms concerning its multiplication operator. More precisely, these axioms concern the closure of S under multiplication, the associativity of multiplication, the existence of an identity element in S with respect to multiplication, and the distributivity of multiplication over addition. Indeed, the former three axioms are analogs of the above-defined "Closure", "Associativity", and "Identity Element" axioms; contrarily, the distributivity axiom is entirely novel and states the following. Here, we denote the multiplication operator by $\cdot$.

- *Distributivity*
  For all $x, y, z \in S$, we have $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$ and $(x + y) \cdot z = (x \cdot z) + (y \cdot z)$.

Furthermore, if the ring's multiplication operator satisfies the cognate of the aforementioned "Commutativity" axiom, the ring is termed a *commutative ring*.

Finally, an *algebraic structure* is a non-empty set equipped with a (potentially empty) set of operators and a finite set of axioms defining the behavior of these operators; certainly, groups and rings are examples of algebraic structures.

Univariate Polynomials

Univariate Polynomials are mathematical expressions consisting of a sum of terms, where each term is constructed as the multiplication of a *coefficient* and a non-negative power of the considered *indeterminate*. Here, the coefficients represent constants from a particular commutative ring, while the indeterminate signifies an abstract value from the same commutative ring[1]. Denoting the coefficients by a_i and the indeterminate by X, we can represent any univariate polynomial $a(X)$ as follows.

$$a(X) = \sum_{i=0}^{\infty} a_i \cdot X^i$$

This summation can be shortened based on the *degree* of $a(X)$, which equals the greatest i for which $a_i \neq 0$, where 0 signifies the ring's identity element with respect to addition. Specifically, assuming the degree of $a(X)$ equals n, the above can be rewritten as given below

$$a(X) = \sum_{i=0}^{n} a_i \cdot X^i$$

The addition and multiplication operators for univariate polynomials are straightforwardly defined based on the (properties of the) analogous operations of the underlying ring.

Crucially, cognates of the previously introduced Euclidean division and modulo operation exist in the context of univariate polynomials. In particular, given two polynomials $a(X)$ and $b(X)$,

[1]Technically, polynomials may be defined over different types of algebraic structures as well; however, since this thesis only considers polynomials over commutative rings, such generalization is gratuitous.

where $b(X)$ has a unit as its leading coefficient, there exists unique polynomials $q(X)$ and $r(X)$ such that either $r(X) = 0$ or $\deg(r(X)) < \deg(b(X))$, and the following equation holds[2].

$$a(X) = q(X) \cdot b(X) + r(X)$$

Correspondingly, considering the same context, the imminent equation concerning the modulo operation is veracious.

$$r(X) = a(X) \bmod b(X)$$

In consequence of this Euclidean division for polynomials, the aforementioned concept of modular arithmetic also exists in the context of univariate polynomials. Namely, in this context, two polynomials $a(X)$ and $b(X)$ are congruent with respect to a (polynomial) modulus $p(X)$ if $a(X) \bmod p(X) = b(X) \bmod p(X)$. As with the integers, grouping the polynomials based on this congruence relation produces a finite number of distinct sets of polynomials; the union of these sets comprises all polynomials. Moreover, computations in such a system are carried out analogously to the computations in a modular arithmetic system for integers; i.e., computations are performed on the considered congruence classes, which practically entails replacing each congruence class in a computation with an arbitrary polynomial from this class and, afterward, operating on these polynomials.

Lastly, although "polynomials" is an umbrella term encompassing both univariate and multivariate polynomials, we use "polynomials" synonymously with "univariate polynomials" throughout this thesis. Indeed, since we solely consider univariate polynomials, this should not cause any ambiguity.

Vectors and Matrices

A *matrix* is a two-dimensional array with a fixed number of rows and columns; at each intersection of these rows and columns, the matrix contains an *entry*, i.e., an element of a commutative ring [3]. If a matrix has m rows and n columns, we say that the matrix has dimension $m \times n$; additionally, in the special case that $m = 1$ or $n = 1$, the matrix may also be referred to as a n-dimensional *row vector* or a m-dimensional *column vector*, respectively. At times where it is either evident or irrelevant whether the considered object is a row vector or a column vector, we will frequently omit "row" or "column".

Regarding matrices and vectors, several fundamental operators are of importance for the work in this thesis. First, a matrix's rows and columns may be swapped in a pair-wise manner through the *transpose* operator, denoted by $\cdot^T$; more precisely, applying this operator to a matrix swaps the first row with the first column, the second row with the second column, et cetera. Second, matrix addition is only well-defined for matrices with an identical dimension; the addition of two such matrices produces the matrix that results from entry-wise addition. Third, a matrix can be multiplied by a single element from their entry ring; this multiplication is denominated *scalar multiplication* and produces the matrix that is obtained from individually multiplying each of the original matrix's entries by the single element. At last, matrix multiplication is solely well-defined if the number of columns of the left-hand side matrix is equal to the number of rows of the right-hand side matrix. If this is the case, the entry at the i-th row and the j-th column of the resulting matrix is computed by multiplying each entry in the i-th row of the left-hand side matrix by the corresponding entry in the j-th column of the right-hand side matrix and, subsequently, summing these entry-wise products. Indeed, the number of rows of the resulting matrix equals that of the left-hand side matrix; similarly, the number of columns of the resulting matrix is identical to that of the right-hand side matrix.

[2]A unit is an element of a ring that has an inverse with respect to the ring's multiplication operator. Furthermore, $\deg(\cdot)$ denotes, as its name suggests, the degree of its argument.

[3]As with polynomials, matrices can technically be defined over algebraic structures other than commutative rings; nevertheless, again, because this thesis solely regards matrices over commutative rings, such generalization is unnecessary.

Structures of Interest

Having recapitulated the relevant rudimentary concepts from (abstract) algebra and number theory, we introduce the algebraic structures that are of particular interest for this thesis.

First, the (commutative) ring of integers modulo n, where n is an integer greater than 1, constitutes the most fundamental algebraic structure ubiquitously utilized throughout this thesis. This ring precisely corresponds to the above-discussed modular arithmetic system for integers with respect to modulus n. That is, the ring's set comprises the congruence classes of this modular arithmetic system; furthermore, the ring's addition and multiplication operator are carried out on the congruence classes, as in the modular arithmetic system. Indeed, the combination of this set with these operators satisfies the commutative ring axioms as a direct consequence of the fact that the set of integers with integer addition and multiplication conforms to these same axioms.

Second, analogous to the ring of integers modulo n, the (commutative) ring of polynomials modulo $p(X)$, where $p(X)$ is a polynomial with a unit as its leading coefficient, exactly corresponds to the aforementioned modular arithmetic system for polynomials with respect to $p(X)$. Furthermore, we exclusively consider such polynomial rings concerning polynomials over the ring of integers modulo n.

Unless explicitly stated otherwise, in computations concerning elements from instances of the two above-introduced rings, we assume *canonical representatives* of the congruence classes. Essentially, the canonical representative of a congruence class is the remainder obtained through Euclidean division of any element within that congruence class; by definition, such a representative is itself included in the congruence class. Certainly, these canonical representatives are, in the case of integers, the (unique) smallest element and, in the case of polynomials, the (unique) lowest-degree element of their respective congruence classes. Regarding a ring of integers modulo n, this implies that each element is represented by an integer in the range $[0, n-1]$. Moreover, for a ring of polynomials modulo $p(X)$, this validates the assumption that each element is represented by a polynomial with a degree strictly less than the degree of $p(X)$.

Finally, we frequently utilize groups of $m \times n$-dimensional matrices over a particular commutative ring; in most cases, this commutative ring constitutes an instance of the above-mentioned ring of polynomials modulo $p(X)$.

2.1.2 Probability Theory

Albeit less ubiquitously than the topics from abstract algebra and number theory, we occasionally employ concepts from the field of probability theory; these concepts primarily regard particular probability computations and distributions. Presently, we succinctly summarize the relevant material related to these concepts. Here, we predominantly provide intuitive descriptions and definitions, abstracting away from unnecessary mathematical granularity and technicalities; this should be adequate for the work in this thesis.

Probability Computations

In probability theory, a *random experiment* is a repeatable procedure with a well-defined set of distinct possible outcomes, the *sample space*. If this sample space is discrete, which is invariably the case throughout this thesis, each outcome can be associated with a particular probability that denotes the likelihood of the outcome happening; axiomatically, this probability must be a value in the continuous range $[0, 1]$ and the probabilities of all possible outcomes must precisely sum to 1. Combining several possible outcomes, an *event* is any subset of the sample space[4]. An event has *occurred* if the outcome of the random experiment is contained within the event.

[4]Indeed, an event can also be a singleton set comprising a single possible outcome.

Correspondingly, the probability of an event's occurrence is defined as the sum of the probabilities associated with the outcomes encompassed by the event. Two events are *mutually exclusive* if their intersection equals the empty set, i.e., the events do not share any mutual outcomes; two events are *independent* if they do not influence each other's probability of occurring. Regarding the probability computations in this thesis, we exclusively consider mutually exclusive events.

Given n mutually exclusive events A_i, where $1 \leq i \leq n$, with associated probabilities $\Pr[A_i]$, the following identity defines the probability of any of these events occurring.

$$\Pr\left[\bigcup_{i=1}^{n} A_i\right] = \sum_{i=1}^{n} \Pr[A_i]$$

Probability Distributions

Intuitively, a *probability distribution*, or *distribution*, is a mathematical function that defines the probability of occurrence for each event considered in a random experiment. If the set of these events is countable, then the probability distribution is *discrete*. In this thesis, we merely conduct one type of random experiment called *sampling*: the random extraction of a single element from a set. This random experiment exclusively considers all singleton events, each comprising a single possible outcome. As such, the probability of extracting a specific element in this random experiment is equal to the probability of occurrence associated with the corresponding (singleton) event; certainly, this is contingent on the considered distribution. In total, we employ two distributions: the (discrete) uniform distribution and the centered-binomial distribution; both of these distributions are discrete.

The *discrete uniform distribution* assigns an equal probability of occurrence to each possible outcome. As such, the probability of obtaining a specific element by sampling from this distribution is equal for all elements in the considered (countable) set.

The *centered binomial distribution* is a discrete probability distribution designed to approximate a discrete Gaussian distribution while simultaneously facilitating efficient and secure implementations in software [32]. Parameterized by a strictly positive integer μ, the centered binomial distribution considers, by definition, the integer range $[-\mu, -\mu+1, \dots, \mu-1, \mu]$ as the (countable) set of possible outcomes; furthermore, the probability of acquiring a specific element k from this range by sampling from the centered binomial distribution with parameter μ is defined by the following function [33].

$$f_\mu(k) = \frac{(2 \cdot \mu)!}{(\mu + k)! \cdot (\mu - k)!} \cdot (\frac{1}{2})^{2 \cdot \mu}$$

2.2 Cryptography

In this section, the covered material without locally provided references can, with the exception of the material concerning lattice-based cryptography, be found in the majority of fundamental cryptography books, e.g., [1] and [34]. For the material regarding lattice-based cryptography, there exist slightly more dedicated books such as [35].

2.2.1 Public-Key Cryptography

As indicated in Chapter 1, this thesis analyzes Saber's PKE scheme, the core component of Saber's KEM. Both of these cryptographic constructions belong to the category of *public-key cryptography*, i.e., they based on public-key primitives. In the imminent, we elaborate on the general compositions of such schemes.

Public-Key Encryption Scheme

A *public-key encryption scheme*, or PKE scheme, is a cryptographic construction enabling any entity A to encrypt a message into a ciphertext for a specific entity B such that solely B can decrypt this ciphertext and, hence, access the original message. Here, the encryption process is carried out with B's public key; the decryption process is performed with B's private key (or secret key). Indeed, these keys correspond to the public-key primitive on which the PKE scheme is based. In particular, as addressed in Chapter 1, this implies that the public key may be distributed to any entity, while the secret key must remain exclusively known to B. Furthermore, the security of the PKE scheme is entirely reliant on the hardness of the mathematical problem that underlies the scheme's operations and the relationship between the keys.

Formally, a PKE scheme is a triple of algorithms that comprises a key generation algorithm, an encryption algorithm, and a decryption algorithm. Furthermore, each PKE scheme defines several domains: a *key space*, a *message space*, and a *ciphertext space*[5]. As its name suggests, the former represents the domain of valid key pairs, i.e., (public key, private key) tuples; similarly, the latter two respectively denote the domain of valid messages and ciphertexts. Given any $\text{PKE} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$, key space $\mathcal{K}$, message space $\mathcal{M}$, and ciphertext space $\mathcal{C}$, the general specification of the scheme is as follows.

- **Key Generation** $\mathsf{KeyGen}()$
  Input: None.

  Output: A key pair $(\mathsf{pk}, \mathsf{sk})$ from key space $\mathcal{K}$.

- **Encryption** $\mathsf{Enc}(m, \mathsf{pk})$
  Input: A message m from message space $\mathcal{M}$ and, generated in accordance with KeyGen, a public key pk.

  Output: A ciphertext c, the encryption of m.

- **Decryption** $\mathsf{Dec}(c, \mathsf{sk})$
  Input: A ciphertext c from ciphertext space $\mathcal{C}$ and, generated in accordance with KeyGen, a secret key sk.

  Output: A message m', the decryption of c, or $\perp$, the explicit indication of a decryption failure.

Naturally, a PKE scheme is perfectly correct if for any $m \in \mathcal{M}$ and $(\mathsf{pk}, \mathsf{sk})$ generated by $\mathsf{KeyGen}()$, we have $\mathsf{Dec}(\mathsf{Enc}(m, \mathsf{pk}), \mathsf{sk}) = m$.

Albeit not explicitly indicated above, a PKE scheme operates with respect to some security parameter. This security parameter determines the level of the scheme's security; typically, this security level manifests itself in the length of the keys generated by the key generation algorithm.

Key Encapsulation Mechanism

A *key encapsulation mechanism*, or KEM, is a cryptographic construction that allows any entity A to generate and *encapsulate* a symmetric key for a specific entity B such that only B can *decapsulate* the encapsulation and, thus, access the symmetric key[6]. Here, the process of generating and encapsulating a symmetric key, i.e., encapsulation, is monolithic; that is, the encapsulation

[5]Frequently, one or more of these domains are defined implicitly through the specifications of the scheme's algorithms.

[6]A symmetric (secret) key is utilized in secret-key primitives which, in contrast to public-key primitives, use the same key for both encryption and decryption.

procedure invariably produces both a symmetric key and the corresponding encapsulation. Analogous to the encryption and decryption procedures of a PKE scheme, the KEM's encapsulation and decapsulation procedures are respectively carried out using B's public key and B's private key.

Formally, a KEM is a triple of algorithms that comprises a key generation algorithm, an encapsulation algorithm, and a decapsulation algorithm. Moreover, each KEM defines several domains: an *asymmetric key space*, a *symmetric key space*, and an *encapsulation space*[7]. Concerning these domains, the former is directly analogous to the key space for PKE schemes; furthermore, the latter two represent the domains of valid symmetric keys and encapsulations, respectively. For any KEM $= (\mathsf{KeyGen}, \mathsf{Encaps}, \mathsf{Decaps})$, asymmetric key space $\mathcal{AK}$, symmetric key space $\mathcal{K}$, and encapsulation space $\mathcal{C}$, the general specification of the scheme is as follows.

- **Key Generation** $\mathsf{KeyGen}()$
  Input: None.

  Output: A key pair $(\mathsf{pk}, \mathsf{sk})$ from key space $\mathcal{AK}$.

- **Encapsulation** $\mathsf{Encaps}(\mathsf{pk})$
  Input: A public key pk generated in accordance with KeyGen.

  Output: A tuple (k, c) where k is a symmetric key and c is the encapsulation of k.

- **Decapsulation** $\mathsf{Decaps}(c, \mathsf{sk})$
  Input: A ciphertext c from encapsulation space $\mathcal{C}$ and, generated in accordance with KeyGen, a secret key sk.

  Output: A symmetric key k', the decapsulation of c, or $\perp$, the explicit indication of a decapsulation failure.

Evidently, a KEM is perfectly correct if for any $(\mathsf{pk}, \mathsf{sk})$ generated by $\mathsf{KeyGen}()$ and (k, c) produced by $\mathsf{Encaps}(\mathsf{pk})$, we have $\mathsf{Decaps}(c, \mathsf{sk}) = \mathsf{k}$.

Although not explicitly denoted above, a KEM operates with respect to some security parameter that determines the level of the scheme's security; usually, this security level manifests itself in the length of the keys generated by the key generation algorithm.

2.2.2 Provable Security

In cryptography, *provable security* is a paradigm used to formally argue for the supposed security of cryptographic constructions. This paradigm relies on the formal definition of cryptographic constructions, security properties, relevant adversaries, and potential hardness assumptions; certainly, the latter includes assumptions on the hardness of specific mathematical problems underlying the considered cryptographic construction, a concept previously mentioned in Chapter 1. After their formal definition, these artifacts are utilized in a rigorous proof showing that, provided the hardness assumptions are veracious, the cryptographic construction possesses a certain security property, i.e., the security of the construction cannot feasibly be compromised by any relevant adversary. Generally, the set of "relevant adversaries" exclusively comprises adversaries sufficiently restricted in their resource utilization; for instance, this set is frequently defined as the set of all probabilistic, polynomial-time algorithms. The rationale for this is the confinement of the analysis to practically relevant and meaningful scenarios; particularly, hardness assumptions are invariably contingent

[7] As for the analogous domains in the context of PKE schemes, these domains are often implicitly defined through the specifications of the scheme's algorithms.

on the fact that certain problems cannot be solved efficiently (with sufficient success probability). Here, "solved efficiently" essentially denotes "solved by a certain set of resource-restrained adversaries". Indeed, hardness assumptions are trivially invalid in the context of adversaries with access to unlimited resources.

Multiple approaches to the provable security paradigm exist and, depending on the considered context, some approaches might be more suitable than others [36, 37]. In this thesis, we employ an approach in which security properties and hardness assumptions are formalized as probabilistic programs defined with respect to an arbitrary relevant adversary; these programs are denominated *games* and have a well-defined syntax and semantics. In these games, the adversary is tasked with solving a particular problem: in the games modeling a security property, solving this problem relates to compromising security; in the games formalizing a hardness assumption, solving this problem relates to finding a solution to the underlying mathematical problem. More precisely, the games are formulated in such a way that, if an adversary is capable of successfully solving the presented problem with a greater probability than what is trivially achievable, then this excess probability must originate from information obtained by, for security properties, (partially) compromising security or, for hardness assumptions, (partially) solving the underlying mathematical problem. As a convention, correctly solving the problem presented in a game is referred to as *winning* the game; furthermore, the difference between the trivially achievable winning probability and the actual winning probability of an adversary is termed the *advantage* of that adversary against the considered game.

Befittingly, the type of proof corresponding to the above-introduced games is called a *game-based proof* or *game-playing proof*; occasionally, to accentuate the fact that games are modeled as probabilistic programs, this type of proof is referred to as a *code-based, game-playing proof.* In such a proof, we show that a particular cryptographic construction satisfies the desired security property by formally manipulating the corresponding game, i.e., the game that formalizes this security property with respect to the considered construction. For each performed manipulation, we provide a justification; more precisely, we formally argue that one of the following conditions is satisfied[8]. Here, "original game" and "resulting game" refer to the game on which the manipulation is performed and the game resulting from the manipulation, respectively.

- Winning the original game is at least as hard as winning the resulting game.
- The probability of a relevant adversary differentiating between (the problem presented in) the original game and (the problem presented in) the resulting game is negligibly small[9]. This type of justification might be based on the veracity of the hardness assumption(s).

Oftentimes, these justifications are formalized by means of a *reduction* which, given a relevant adversary against one or multiple games, constructs a relevant adversary against another game. For example, regarding the first type of justification in the list above, a reduction may utilize a relevant adversary against the original game to create a relevant adversary against the resulting game such that, between both these adversaries, the probability of winning their respective games is identical; indeed, this implies that winning the original game is at least as hard as winning the resulting game.

Ultimately, after performing a series of manipulations, we intend to reach a game for which we can prove that no relevant adversary can win with a probability greater than what is trivially achiev-

[8]This is not intended to be an exhaustive list of potential justifications allowed in game-playing proofs; instead, this list comprises some of the most customary types of justifications, including all types utilized throughout this thesis.

[9]Technically, "negligibly small" is related to the mathematical definition of a *negligible function*; nevertheless, for the discussion throughout this thesis, the linguistic interpretation of this phrase suffices and, hence, we disregard the corresponding mathematical formality.

able. At this point, employing the sequence of formal justifications for the imposed manipulations, we can bound the advantage of any relevant adversary against the initial game, i.e., the game representing the desired security property of the considered construction, from above. In particular, this bound is expressed in terms of the advantages of several other relevant adversaries against the games modeling the hardness assumption(s). If the hardness assumptions are veracious, these advantages are negligibly small irrespective of the considered adversaries; in turn, the established bound and, hence, the advantage of any relevant adversary against the initial game are negligibly small as well. Concluding, this proves that, in the context of the considered construction, no relevant adversary can sufficiently compromise the desired security property, implying that the considered construction indeed possesses this property.

2.2.3 Security Properties

Hitherto, we have primarily discussed the security of cryptographic constructions in an abstract manner. Nevertheless, in the introduction of Saber, see Chapter 1, we briefly mentioned the concrete security properties that the schemes of Saber intend to achieve. To reiterate, for Saber's PKE scheme, this property is ciphertext indistinguishability under chosen-plaintext attack. In the following, we describe this property in more detail.

Indistinguishability under Chosen Plaintext Attack

Ciphertext *indistinguishability under chosen-plaintext attack*, or IND-CPA security, refers to a security property commonly desired from cryptographic constructions that intend to provide confidentiality, e.g., PKE schemes. Intuitively, the notion of IND-CPA security states that no relevant adversary should be feasibly capable of distinguishing ciphertexts based on the messages they encrypt; naturally, these ciphertexts are to be constructed per the considered construction's specification. Albeit IND-CPA security can apply to several classes of confidentiality providing cryptographic constructions, we restrict the explication of this property to the context of PKE schemes. This is because, in this thesis, we exclusively consider this property with respect to a PKE scheme, viz., Saber's PKE scheme.

Considering a particular PKE scheme, the IND-CPA security property is generally modeled by the following game-like scenario, defined with respect to a relevant adversary. First, a key pair is generated per the PKE scheme's key generation algorithm. Then, provided with the generated public key, the adversary selects two messages from the considered message space. Subsequently, chosen uniformly at random and unbeknownst to the adversary, one of these messages is encrypted with the PKE scheme's encryption algorithm. Lastly, given the ciphertext resulting from this encryption, the adversary attempts to determine which of the two messages was encrypted to obtain this ciphertext. Based on this scenario, the PKE scheme is considered to be IND-CPA secure if no relevant adversary can successfully make the final determination with a probability that is non-negligibly different from $\frac{1}{2}$. Here, the adversary's success probability is required to be non-negligibly different from $\frac{1}{2}$ because a success probability of $\frac{1}{2}$ is trivially achievable. As such, the difference between the attained success probability and $\frac{1}{2}$ is what actually captures the extent to which the adversary could extract information about the message from the ciphertext.

In the above scenario, the considered adversary is not limited in any regard apart from the general restrictions imposed on the class of relevant adversaries. In particular, this implies that, within the boundaries set by these imposed restrictions, the adversary is allowed to arbitrarily compute and store data, including the encryption of arbitrary messages through the use of the provided public key and the PKE scheme's encryption algorithm[10]. Moreover, by allowing the adversary

[10]This insinuates that, in order for a PKE scheme to be IND-CPA secure, its encryption algorithm must be probabilistic; that is, repeatedly encrypting the same message with the same public key should result in different ciphertexts. Certainly, if this is not the case, the adversary can straightforwardly determine the message corresponding to the provided ciphertext by encrypting both candidate messages and, subsequently, verifying which of

to pick the messages itself, the scenario excludes the possibility of making the adversary's success probability dependent on the particular structure of the messages. As such, the scenario indeed models the property that no relevant adversary should be capable of distinguishing ciphertexts based on the messages they encrypt.

From the preceding, it is evident that IND-CPA security relates to the confidentiality that the considered PKE scheme provides; however, the extent to which it does is less apparent. Nevertheless, if modeled in accordance with the above scenario, IND-CPA security has been proved equivalent to *semantic security under chosen-plaintext attack*. That is, no relevant adversary can feasibly extract more than negligible information about a message from its encryption.

2.2.4 Lattice-Based Cryptography

Akin to groups and rings, *lattices* are algebraic structures; in fact, they constitute a specific class of groups. Intuitively, lattices represent an infinite collection of evenly spread out vertices in a multidimensional space. Utilizing these algebraic structures, *lattice-based cryptography* considers cryptographic constructions contingent on hardness assumptions that are related to computational problems in lattices; indeed, Saber's schemes belong to this category. Crucially, according to current understanding, the computational problems utilized for lattice-based cryptography are, even considering quantum computation, not efficiently solvable. In the ensuing, we introduce the computational problems and hardness assumptions relevant to Saber. However, since the material related to these topics is quite mathematically intricate and beyond the scope of this thesis, we primarily restrict the discussion to intuitive descriptions and definitions.

(Gap) Shortest Vector Problem and Shortest Independent Vector Problem

The *Shortest Vector Problem (SVP)* is among the most prevalent computational problems for lattices. Intuitively, given a particular lattice, this problem concerns finding the shortest non-zero vector in the lattice. A decisional version of this problem, the *Gap Shortest Vector Problem (GapSVP)*, involves deciding whether the shortest non-zero vector in a certain lattice is shorter than some length d. Comparable to SVP, the *Shortest Independent Vector Problem (SIVP)* entails finding a specific number of linearly independent lattice vectors such that the length of the longest among these vectors is minimal. For each of these problems, any *norm* may be used as a measure of vector length, although the *Euclidean norm* is the most common. Typically, within cryptography, computational complexity analyses consider *approximation* variants of the above-introduced *exact* SVP, GapSVP and SIVP problems. Indeed, rather than the exact solution, these variants involve finding an approximate solution to the problems. Informally, these approximation variants are defined as follows.

- SVP_γ: Given a lattice, find a lattice vector with a length that is within a factor γ from the length of the shortest lattice vector.
- GapSVP_γ: Given a lattice and length d, decide whether the shortest lattice vector is shorter than $\gamma \cdot d$.
- SIVP_γ: Given a lattice, find a specific number of linearly independent lattice vectors such that the length of the longest among these vectors is within a factor γ from minimal.

At present, if γ is taken to be a polynomial in the rank of the lattice, no algorithm, classical or quantum, exists for solving these problems efficiently.

the encrypted messages is identical to the provided ciphertext.

(Module-)Learning With Errors and (Module-)Learning With Rounding

Related to several of the aforementioned computational problems in lattices, the *Learning With Errors (LWE)* problem and its variants constitute the basis of numerous contemporary post-quantum cryptographic constructions [38,39]. Parameterized by a distribution χ over the integers and two strictly positive integers n and q, the LWE problem is based on a specific distribution; this distribution is uniquely determined by $\mathbf{s}$, a n-dimensional vector over the ring of integers modulo q, and χ [40]. Sampling from this distribution is accomplished through the following procedure [39].

1. Sample $\mathbf{a}$ uniformly at random from the set of n-dimensional vectors over the ring of integers modulo q.

2. Subsequently, sample e from χ and compute $b = (\mathbf{a}^T \cdot \mathbf{s} + e) \bmod q$. Here, b is interpreted as an element from the ring of integers modulo q.

3. At last, output the tuple $(\mathbf{a}, b)$.

Customarily, this distribution and its samples are referred to as "LWE distribution" and "LWE samples", respectively. Furthermore, the contexts that employ the LWE distribution oftentimes require multiple, say m, LWE samples; this includes the context of Saber. In these contexts, we can leverage techniques from linear algebra to efficiently denote and perform the sampling procedure. Namely, first, we sample $\mathbf{A}$ uniformly at random from the set of $m \times n$-dimensional matrices over the ring of integers modulo q; certainly, each row of such a matrix constitutes a uniformly random n-dimensional vector over the same ring, i.e., a LWE sample. Second, we sample $\mathbf{e}$ from χ^m and compute $\mathbf{b} = \mathbf{A} \cdot \mathbf{s} + \mathbf{e}$. Here, χ^m denotes the distribution over m-dimensional vectors for which the sampling procedure is equivalent to independently sampling each of the m vector entries from χ; furthermore, each entry of $\mathbf{b}$ is interpreted as an element from the ring of integers modulo q. By the definition of matrix multiplication, carrying out $\mathbf{A} \cdot \mathbf{s} + \mathbf{e}$ is identical to computing $\mathbf{a}_i^T \cdot \mathbf{s} + e_i$ for $0 \leq i \leq m$, where $\mathbf{a}_i$ and e_i respectively signify the i-th row of $\mathbf{A}$ and i-th entry of $\mathbf{e}$. Lastly, for the third step, we output the tuple $(\mathbf{A}, \mathbf{b})$, essentially comprising m LWE samples; specifically, each pair $(\mathbf{a}_i, b_i)$, $0 \leq i \leq m$, constitutes a single LWE sample.

Utilizing the above distribution, the search version of the LWE problem, *search LWE*, involves finding $\mathbf{s}$ based on a particular number of given LWE samples; the decisional version of this problem, *decision LWE*, concerns distinguishing the given LWE samples from the same number of uniformly random samples over the same domain [39]. Intuitively, these problems are hard due to the addition of e to the matrix multiplication $\mathbf{a}^T \cdot \mathbf{s}$; indeed, without the addition of this error, i.e., when $e = 0$, the problems are trivial to solve [40]. In the quantum setting, considering a suitable q, χ, and number of samples, both versions of the LWE problem have been shown to be at least as hard as the GapSVP_γ and SIVP_γ problems on arbitrary n-dimensional (full-rank) lattices; in the classical setting, a nearly identical result has been proved, yet merely with respect to GapSVP_γ [40–42]. Notably, in either setting, a suitable choice for χ is the discrete Gaussian distribution [39].

Although the LWE problem is endowed with promising hardness proofs, the utilization of this problem in cryptographic constructions inherently induces a considerable overhead and, hence, significantly reduces the constructions' efficiency [43]. In an attempt to reduce this problematic overhead while preserving sufficient computational complexity, variants of the LWE problem were developed. An initial such variant constitutes the *Ring-Learning With Errors (RLWE)* problem. The only discrepancy between the versions of this variant and their LWE counterparts is that, in places where the latter utilize (n-dimensional vectors with) elements of the ring of integers modulo q, the former employ polynomials of degree at most n from a certain polynomial ring; as a result of this difference, each RLWE sample can encode as much information as n LWE samples [43].

This substantially reduces the key sizes required in cryptographic constructions, which is the predominant performance issue with constructions based on versions of the LWE problem [38, 43]. However, due to the structure leveraged by the versions of the RLWE problem, they can solely be proved at least as hard as some of the above-discussed lattice problems in arbitrary *ideal* lattices, a specific class of lattices [43, 44]. These “ideal lattice problems” have been studied to a much lesser extent than their general counterparts; moreover, particular problems that are hard in general lattices are known to be easy in ideal lattices, e.g., the GapSVP_γ problem [38, 45]. For these reasons, the *Module-Learning With Errors (MLWE)* problem emerged, striking a balance between the LWE and RLWE problems [46]. The versions of the MLWE problem are nearly identical to their RLWE cognates, merely replacing certain individual polynomial elements with d-dimensional vectors of such elements, where d is an additional parameter of the problem; indeed, this implies that the versions of the MLWE problem with $d = 1$ are identical to their RLWE analogs [45, 47]. As with the RLWE problem, the versions of the MLWE problem can only be shown at least as hard as particular computational problems in a specific class of lattices; in this case, this is the class of *module* lattices. Due to the different, more complicated structure of these lattices relative to ideal lattices, the strategies for solving some of the computational problems in ideal lattices are not applicable; nevertheless, when utilized as the underlying mathematical problem for cryptographic constructions, the versions of the MLWE problem still admit smaller key sizes than their LWE counterparts [46, 47]. As an additional benefit, building cryptographic constructions based on the MLWE problem allows for greater flexibility in the trade-off between efficiency and security [23, 47].

As previously mentioned in Chapter 1, Saber is based on the MLWR problem, a variant of the LWR problem [14]. As their names suggest, these problems are closely related to the MLWE problem and the LWE problem, respectively. Namely, the sole difference between the LWE and LWR problems manifests itself in the procedure of sampling from their respective distributions [15]. More precisely, instead of introducing an error to the matrix multiplication $\mathbf{a}^T \cdot \mathbf{s}$ via the addition of a randomly sampled error e, this error is produced by scaling the matrix multiplication by a factor $0 < \frac{p}{q} \leq 1$ and, subsequently, rounding the outcome to the nearest integer. The result of this scaling and rounding is interpreted as an element from the ring of integers modulo p (rather than q); as a result, the LWR problem is defined with respect to a (strictly) positive integer parameter p for which $p \leq q$ [15]. Furthermore, the relation between the LWR and MLWR problems is analogous to the relation between the LWE and MLWE problems [14]. Lastly, for suitable parameters, both of the LWR and MLWR problems have respectively been shown to be at least as hard as the LWE and MLWE problems [15, 16, 48]

2.2.5 Random Oracle Model

As indicated in the discussion on provable security, cryptographic proofs may rely on several assumptions. These assumptions can, for instance, conjecture the intractability of a particular problem or limit the set of relevant adversaries by imposing restrictions on the capabilities of an adversary. Cryptographic proofs that solely depend on these types of assumptions are said to be staged in the *standard model.* Nevertheless, frequently, cryptographic proofs additionally utilize another type of assumption; namely, they often assume that every function aimed at generating *pseudorandomness*, i.e., data that is intended to be computationally indistinguishable from truly random data, is a *random oracle.* A random oracle is a mathematical function selected uniformly at random from the set of all mathematical functions defined on a specific domain and codomain; this function selection is equivalent to independently mapping each element from the domain to an element from the codomain in a uniformly random manner. Suitably, cryptographic proofs that employ this random oracle assumption are said to be staged in the *Random Oracle Model (ROM).* In such proofs, random oracles are modeled as distinct black-box entities accessible by all other entities considered in the proof. Accessing a random oracle constitutes querying the oracle on some element from its domain, upon which the oracle (efficiently) computes and returns the image of this value. Indeed, as directly implied by the definition of a random oracle, each domain element is invariably mapped to the same codomain element; furthermore, the output distribution of a random oracle equals the uniform distribution over the codomain.

The rationale for performing cryptographic proofs in the ROM rather than the standard model arises from the fact that most cryptographic constructions require the use of replicable randomness. That is, these constructions require mathematical functions of which the output distribution is uniformly distributed over the codomain; indeed, such a function is a random oracle. Unfortunately, it is impossible to practically implement a random oracle efficiently, i.e., in polynomial space and time complexity; in turn, cryptographic constructions cannot utilize random oracles without becoming impossible to implement efficiently as well. Alternatively stated, if a cryptographic construction is ever to be practically implemented and deployed, it cannot employ random oracles. For this reason, as suggested above, contemporary cryptography uses functions, such as particular hash functions, that generate pseudorandomness instead of true (replicable) randomness. Since the output of such functions is not truly random, the desired properties of cryptographic constructions might be violated by virtue of the specific functions utilized to generate the necessary pseudorandomness; consequently, formally reasoning about these properties becomes quite arduous. However, modeling these pseudorandomness-generating functions as random oracles, i.e., staging the proofs in the ROM, eliminates all their potential flaws, facilitating the formal validation of the overall cryptographic construction.

2.3 Computer-Aided Cryptography

2.3.1 EasyCrypt

As mentioned in the preceding chapter, EasyCrypt is the tool of choice for the computer-aided verification effort effectuated in this thesis. Considering the context of this verification effort, we presently discuss the conceptual approach, relevant features, and basic usage of this tool. In this discussion, we remain relatively pragmatic because, as with several other subjects discussed throughout this chapter, the underlying mathematical theory is quite esoteric and, as such, surpasses the scope of this thesis. Nevertheless, the covered material should be sufficient to readily comprehend the discussion on the actual formal verification effort presented in Chapter 4.

Main Features and Approach

Albeit EasyCrypt's vast set of features gives rise to numerous capabilities with considerable flexibility, the tool's principal objective is to facilitate the formal verification of cryptographic proofs that follow the previously discussed approach to the provable security paradigm; that is, EasyCrypt predominantly intends to provide support for the formal verification of code-based, game-playing cryptographic proofs [18,49]. To this end, the tool accommodates two specification languages with a well-defined syntax and semantics: an *imperative* language, primarily designed for the formal specification of cryptographic constructions, security properties, and hardness assumptions; and a *functional* language, predominantly used for the formal definition of the contexts associated with cryptographic constructions. In addition to these specification languages, EasyCrypt provides an higher-order ambient logic comprising several logics that enable reasoning about (probabilistic) programs or pairs thereof; specifically, these logics include *Hoare Logic (HL)*, *probabilistic Hoare Logic (pHL)*, and *probabilistic Relational Hoare Logic (pRHL)* [50,51].

Utilizing the above features, we can realize the code-based, game-playing approach to the provable security paradigm in the following manner. First, we formalize the relevant context of the considered cryptographic construction in the functional specification language; among others, this entails defining the necessary types and operators. Afterward, using the imperative specification language, we formalize the considered construction, security properties, hardness assumptions, and classes of relevant adversaries. Indeed, in accordance with the employed approach to provable security, we model the security properties and hardness assumptions as probabilistic programs;

hence, they are specified in the imperative language. Moreover, the classes of relevant adversaries are represented by abstract probabilistic programs, i.e., interfaces. This abstract formalization conveys that, apart from the compliance to the input and output requirements, no restrictions are imposed on the behavior of a relevant adversary. Subsequently, we individually formalize each game of the proof's game sequence; that is, in addition to the initial game, every manipulation-induced game in the proof is separately formalized with the imperative language. Completing the collection of proof-related artifacts, we formalize the reductions corresponding to the justifications of the enacted manipulations; certainly, since these reductions are probabilistic programs, this is accomplished by means of the imperative language. At last, using the functional language and ambient logic, we formalize the claims related to the justifications of the manipulations and verify their veracity.

Basic Usage

Concretizing several of the above-mentioned concepts, we illustrate the basic usage of EasyCrypt's relevant features through relatively trivial examples. Simultaneously, these examples serve as an introduction to the notation utilized in the listings throughout this thesis; this notation directly corresponds to EasyCrypt's syntax.

Foremost, we demonstrate the utilization of the functional specification language for several fundamental, customary purposes; most notably, these purposes include the definition of types, operators, axioms, and lemmas. In EasyCrypt, *axioms* and *lemmas* are employed to formalize properties regarding artifacts within the considered script; the difference between these two concepts is the fact that axioms are assumed to be veracious, while lemmas must be proved to be veracious. Listing 2.1 presents the first example; for explanatory purposes, the code in this example is extensively commented[11].

```
(* Define abstract types t and u *)
type t.
type u.

(* Define abstract operators f and g mapping from t to u and from u to t,
    respectively *)
op f : t -> u.
op g : u -> t.

(* State assumptions through axioms - f and g are each others' inverses *)
(* Note - Parameterizing an axiom (or a lemma) is equivalent to universally
    quantifying over the type(s) of the parameter(s) *)
(* So, intuitively, axiom f_inv_g states "For all values x of type u, applying f on
    the image of x under g returns x" *)
axiom f_inv_g (x : u) : f (g x) = x.
axiom g_inv_f (x : t) : g (f x) = x.

(* Define concrete operator h as f composed with g *)
op h (x : u) = f (g x).

(* State to-be-verified properties through lemmas - h is the identity function *)
lemma h_is_id (x : u) : h x = x.
proof. (*...*) qed.

(* Define concrete type p as the set of 2-tuples of integers and booleans *)
type p = int * bool.

```

[11]Comments have the following format: `(* Text of comment *)`.

```
(* Define concrete operator cextr that, given a value of type p, extracts the
    integer (i.e., first element) from this value if its boolean (i.e., second
    element) evaluates to true; else, cextr returns 0 *)
(* Note - "fun (x) => e" denotes an anonymous/lambda function taking an argument x
    and computing expression e *)
(* Note - x.`1 and x.`2 respectively extract the first and second element from a
    tuple x *)
(* Note - "if ... then ... else ..." signifies the ternary operator equivalent to an
    if-else control structure *)
op cextr : p -> int = fun (x : p) => if x.`2 then x.`1 else 0.

(* Define abstract polymorphic operator pmop that maps from an arbitrary domain to
    the integers; i.e., pmop can take an argument of any type and maps it to an
    integer *)
op ['a] pmop : 'a -> int.
```

Listing 2.1: Basic Usage of the Functional Specification Language

Evidently, as indicated by `(*...*)`, the proof of the lemma in this listing is omitted. This is because proving lemmas involves the utilization of EasyCrypt's proof engine and built-in logics, which is quite technically intricate. Considering the more process-oriented nature of the formal verification portion of this thesis's discussion, an elaboration on the details of all of these technical endeavors would not constitute a significant contribution. As such, throughout this thesis, we mostly leave out the concrete proofs of the covered lemmas; instead, we provide an intuitive description of the approaches these proofs employ. Nevertheless, to still give an idea of EasyCrypt's (lemma-)proving process and the corresponding concepts and mechanisms, we ultimately do explicate the proof of one of the lemmas employed in the formal verification endeavor regarding Saber.PKE.

Albeit not mentioned above, an additional foundational feature of the functional specification language regards the specification of *discrete probability sub-distributions*, or *discrete sub-distribution*, over certain types [18, 49]. In contrast to proper discrete distributions, discrete sub-distributions do not guarantee that the sum over all probabilities associated with the values from their domain equals 1; rather, this sum can be any value between 0 (including) and 1 (including) [49]. The functional specification language allows for the creation of such distributions by means of the `distr` *type constructor*; as its name suggests, a type constructor enables the construction of types from other types. Furthermore, due to the commonality of sub-distributions with particular properties, the functional specification language provides a convenient syntax for establishing such sub-distributions. More precisely, this concerns the ensuing properties [18].

- **Lossless**
  The sum of the probabilities associated with all values of the considered type equals 1. That is, the discrete sub-distribution is a proper discrete distribution.

- **Full**
  The probability associated with each value of the considered type is greater than 0. Alternatively stated, sampling from the discrete sub-distribution may yield any value of the considered type.

- **Uniform**
  For some x between 0 (excluding) and 1 (including), the probability associated with any value of the considered type equals either 0 or x.

Listing 2.2 exemplifies the specification of discrete sub-distributions, both with and without the above-mentioned properties.

```
(* Define abstract type t *)
type t.

(* Define dist, a sub-distribution over t *)
op dist : t distr.

(* Define dist1, a lossless sub-distribution over t, i.e., dist1 is a proper
    distribution over t *)
op [lossless] dist1 : t distr.

(* Define dist2, a lossless, full, and uniform sub-distribution over t; that is,
    dist2 is a proper uniform distribution over t, associating all values of type t
    with the same probability (not equal to 0, due to the ''full'' property) *)
op [lossless full uniform] dist2 : t distr.
```

Listing 2.2: Basic Specification of Distributions

As can be extracted from this listing, distributions are defined as operators with only a domain type; that is, these operators have no range type. In EasyCrypt, this is equivalent to specifying a constant of the considered (domain) type. Moreover, if no concrete value is assigned to such an operator, the operator represents an *arbitrary* constant from its (domain) type. Indeed, in case this type corresponds to the type of distributions over some other type, the operator represents an arbitrary constant distribution over this other type. Naturally, despite its arbitrariness, this constant distribution must still adhere to the relevant specified properties, e.g., uniformity.

Next, we illustrate the basic usage of the relevant features from the imperative specification language; particularly, we do so in a manner that accentuates the utilization of these features in the context of a formal verification effort effectuated for a cryptographic proof that follows the previously discussed approach to the provable security paradigm. Listing 2.3 contains the corresponding example, copiously commented for explanatory purposes.

```
(* Context - Define necessary distribution and operator with functional
    specification language *)
op [lossless full uniform] bdist : bool distr.
op f (x : int) = x + 1.

(* Define a module type, i.e., an interface that concrete modules can implement *)
(* In this case, we use the module type feature to represent a particular class of
    adversaries *)
module type Adversary = {
  proc determine(x : int, y : int) : bool
}.

(* Define a (parameterized) module, i.e., an encapsulating artifact comprising
    (often related) probabilistic programs; indeed, each procedure of a module
    constitutes a probabilistic program *)
(* Note - Modules can only be parameterized by modules with a module type;
    subsequently, the encompassing module is able to call the procedures from the
    parameter module that are defined by its module type *)
(* Here, we utilize the module feature to model a (dummy) cryptographic game that
    considers adversaries from the above-defined class (i.e., module type) *)
module Game(A : Adversary) = {
  (* Define a procedure taking a boolean argument and outputting a boolean return
    value *)
  proc main(u : bool) : bool = {
    (* Declare necessary variables *)
    var x, y : int;
```

```
      var u' : bool;
      var b : bool;
      var b1, b2 : bool;

      x <- 1; (* Assign 1 to x *)

      b1 <$ bdist; (* Sample b1 from bdist *)
      b2 <$ bdist; (* Sample b2 from bdist *)

      b <- b1 ^^ b2; (* Assign XOR of b1 and b2 to b *)

      (* if u AND b evaluates to true, ... *)
      if (u /\ b) {
         (* then assign x to y, ... *)
         y <- x;
      } else {
         (* else, assign image of x under f to y *)
         y <- f x;
      }

      (* Call "determine" procedure of parameter module A with arguments x and y *)
      (* Note - Since A has (module) type "Adversary", we know it implements the "determine" procedure *)
      u' <@ A.determine(x, y);

      (* If u' equals u, return true, else return false *)
      return (u' = u);
   }
}.

(* Define a (parameterized) module of a previously defined module type *)
(* Conceptually, the following module models a (dummy) reduction that could be utilized to justify a game step/manipulation in a game-playing cryptographic proof *)
(* Concretely, this module represents an adversary from the above-defined class that employs another adversary from this same class to carry out its procedure *)
module Reduction_Adversary(A : Adversary) : Adversary = {
    (* Implement "determine" procedure to adhere to the "Adversary" module type *)
    proc determine(x : int, y : int) : bool = {
       var u' : bool;

       x <- f y;
       y <- x;

       (* Employ other adversary (i.e., parameter module) *)
       u' <@ A.determine(x, y);

       return u';
    }
}.
```

Listing 2.3: Basic Usage of the Imperative Specification Language

Frequently, games and adversaries in game-playing proofs merely comprise a single algorithm. As such, when reasoning about these algorithms, we generally use the identifier of the game or adversary to refer to them; that is, we do not employ a separate identifier for the algorithm. Nevertheless, the EasyCrypt feature with which games and adversaries are predominantly modeled, i.e., modules, cannot contain concrete, directly executable code themselves; instead, they must define procedures that encapsulate such code for them [18]. Therefore, the module formalizations invariably comprise a dedicated procedure representing the considered algorithm, even when the corresponding conceptual artifacts, e.g., games and adversaries, do not specify a distinct identifier for this algorithm.

Modularization Features

Concerning modularization of formal verification efforts, EasyCrypt provides several enabling features. Certainly, among these are the above-discussed module and modules type, which allow for the modular formalization of cryptographic constructions. Moreover, further facilitating modularization, the *theory* feature permits the encapsulation of one or more related concepts in an abstract, polymorphic manner [18]. For instance, the EasyCrypt library provides a theory that abstractly defines the ring of integers modulo n, for all $1 < n$. In particular, this theory specifies constants, types, operators, axioms, and lemmas which, combined, formalize the fundamental structure of these rings; indeed, this is solely possible due to the fact that this structure is identical across all such rings. Then, by *instantiating*, or *cloning*, this theory with any modulus n that is greater than 1, we essentially define the complete fundamental structure of the corresponding ring of integers modulo n with a single EasyCrypt command. Within an EasyCrypt script, we can instantiate any theory ad infinitum.

Quantum Computation

At the time of writing, EasyCrypt does not yet allow for the consideration of (adversaries capable of) quantum computation [52]. Nevertheless, a currently ongoing project strives to implement features remedying this deficiency; hence, the tool will presumably support the consideration of quantum computation in the near future.

Regarding this thesis, the inability to consider quantum computation does not constitute a hindrance. Namely, the proofs concerning Saber's PKE scheme abstractly define the class of relevant adversaries; in particular, they do not exclude adversaries with access to quantum computation. Therefore, the results of these proofs additionally apply to any relevant adversary that utilizes quantum computation. Nevertheless, the proofs themselves never utilize any quantum-dependent reasoning. As such, the corresponding formal verification effort does not require such reasoning as well.

One caveat to the above exists; namely, the security proof of Saber's PKE scheme is indirectly contingent on ROM proofs regarding the employed hardness assumptions. Generally, elevating ROM proofs to the quantum setting engenders several unique challenges; hence, we cannot assume that, since the proof is correct in the classical setting, it is straightforwardly correct in the quantum setting. However, ROM proofs that solely utilize *history-free* reductions form an exception to this rule [53]. Intuitively, a reduction is history-free if it answers each oracle query independently of any preceding queries. Indeed, the ROM proofs on which the security proof of Saber's PKE relies exclusively employ history-free reductions; as such, this security proof remains valid when lifted to the quantum setting.

2.4 Notation

Disregarding code listings, which utilize a distinct notation and typesetting, the discussion throughout this thesis adheres to several notational guidelines and conventions; specifically, these are the following.

- Identifiers of cryptographic schemes are typeset in roman. For example, Saber.PKE and Saber.KEM could be identifiers of a cryptographic scheme.

- Identifiers of algorithms/procedures and cryptographic key material are typeset in sans serif.

For example, $\mathsf{Saber.KeyGen}$ and $\mathsf{Saber.Enc}$ could be identifiers of an algorithm/procedure; furthermore, pk and sk could be identifiers of a cryptographic key.

- Identifiers of adversaries constitute exactly one capital letter typeset in cursive; moreover, to distinguish between adversaries represented by the same letter, these identifiers may contain subscripts. For example, $\mathcal{A}$ and $\mathcal{B}_1$ could be identifiers of an adversary.

- Identifiers of cryptographic games (in the standard model) invariably adhere to the following format: $\text{Game}^{\rho}_{\mathcal{A},\lambda}$. Here, $\mathcal{A}$ denotes the relevant adversary considered by the game, λ represents the cryptographic construction or the parameter set for which the game is defined, and ρ indicates the property or hardness assumption formalized by the game[12]. In case the game utilizes random oracles, this identifier format changes to $\text{GameROM}^{\rho}_{\mathcal{A},\lambda}$. Analogously, identifiers of probabilistic programs in the standard model and random oracle model respectively conform to the formats $\text{PProg}^{\rho}_{\lambda}$ and $\text{PProgROM}^{\rho}_{\lambda}$, where λ and ρ may denote the same artifacts as before.

- Identifiers of matrix elements constitute exactly one capital letter typeset in boldface. For example, $\mathbf{A}$ and $\mathbf{M}$ could be identifiers of a matrix element.

- Identifiers of vector elements constitute exactly one lowercase letter typeset in boldface. For example, $\mathbf{a}$ and $\mathbf{v}$ could be identifiers of a vector element.

- Identifiers of regular (i.e., non-matrix and non-vector) elements constitute exactly one lowercase letter typeset in italics. For example, a and x could be identifiers of a regular element.

- Identifiers of sets constitute exactly one uppercase letter typeset in italics; furthermore, these identifiers may contain subscripts[13]. For example, X and S_q could be identifiers of a set. Sets that have a conventional notation within the field of mathematics form an exception to this notational guideline. Relevant instances of such sets are the set of natural numbers $\mathbb{N}$, the set of integers $\mathbb{Z}$, and the set of real numbers $\mathbb{R}$.

- By abuse of notation, identifiers of algebraic structures are identical to the identifiers of the sets on which they are defined.

- The binary representation of an unknown or variable integer is denoted by a sequence of lowercase letters typeset in italics; indeed, each letter represents a single bit. Furthermore, to differentiate between individual bits, each letter contains an index, i.e., a number in its subscript. If it is evident from the context which indices are associated with the bits in between the most and least significant bits, these intermediate bits may be replaced by dots. For example, $a_4a_3a_2a_1a_0$ and $b_n \dots b_0$ could be denotations of an integer's binary representation.

- A binary representation of an integer may contain sequences of bits for which the value is known; usually, these bits are replaced by their concrete value. In such binary representations, each sequence of n zero bits is denoted by 0^n; similarly, each sequence of n one bits is denoted by 1^n.

- The uniform distribution over set S is denoted by $\mathcal{U}(S)$.

- The centered binomial distribution with parameter μ is denoted by β_μ. At times, it is convenient to interpret the integer elements on which β_μ is defined as elements from a

[12]Actually, ρ might also be the index of the game in the considered game sequence.

[13]Oftentimes, if present, the subscript denotes some structural property of the set.

different set, e.g., set S; if this is the case, the centered binomial distribution is instead denoted by $\beta_\mu(S)$.

- Sampling from a distribution χ and assigning the result to x is denoted by $x \leftarrow\!\!\$\ \chi$.
- Assigning the evaluation of an expression e to x is denoted by $x \leftarrow e$; likewise, assigning the return value of a call to algorithm/procedure $\mathsf{Proc}(arg_0, \ldots, arg_n)$ to x is denoted by $x \leftarrow \mathsf{Proc}(arg_0, \ldots, arg_n)$.
- The probability of occurrence associated with an event E is denoted by $\Pr[E]$.
- Although PKE is technically an acronym of "public-key encryption", we overload the acronym by occasionally employing it as an abbreviation of "public-key encryption scheme".

Chapter 3

Saber

Prior to engaging in any formal verification effort, it is convenient to have an adequate understanding of the cryptographic constructions in question. Specifically, this is because the formal verification of cryptographic constructions entails formalizing the appropriate specifications and properties; naturally, this formalization process necessitates a sufficient comprehension of the relevant concepts. As such, preceding the chapter that discusses the effectuated formal verification process, this chapter covers the relevant parts of the verified scheme from the Saber cipher suite. In particular, this chapter elaborates on the specification and relevant properties of Saber's PKE scheme; moreover, for these relevant properties, this chapter provides hand-written proofs and analyses that are closely resembled by the formal verification effort discussed in the subsequent chapter.

The remainder of this chapter is structured as follows. First, Section 3.1 covers the preliminaries required to apprehend the ensuing discussion. Afterward, Section 3.2 discusses and analyzes the relevant aspects of Saber's PKE scheme; these aspects include the scheme's specification, security, and correctness.

3.1 Preliminaries

This section goes over the preliminaries necessary to comprehend the subsequent discussion on Saber's PKE. For clarity purposes, these preliminaries are explicated rather minutely, paralleling the meticulousness required for the corresponding formalization process in EasyCrypt.

First, for any natural number $0 < q$, we denote the ring of integers modulo q by $\mathbb{Z}_q$; correspondingly, $\mathbb{Z}_q[X]$ represents the polynomial ring with coefficients in $\mathbb{Z}_q$. As a final extension, we define R_q to be the polynomial quotient ring of $\mathbb{Z}_q[X]$ modulo $X^n + 1$, where n is an integral power of two; that is, $R_q = \mathbb{Z}_q[X]/(X^n + 1)$ in which $n = 2^{\epsilon_n}$ for some $\epsilon_n \in \mathbb{N}$.

Second, for any ring R (e.g., the $\mathbb{Z}_q$, $\mathbb{Z}_q[X]$ or R_q introduced above), we let $R^{m \times n}$ stand for the (additive) group of matrices with dimension $m \times n$ and entries in R. In other words, $R^{m \times n}$ is the group of matrices with m rows, n columns, and where all matrices' entries are elements of R. If either m or n equals 1, $R^{m \times n}$ reduces to the group of, respectively, n-dimensional row vectors or m-dimensional column vectors. In particular, we have $R^{1 \times n} = R^n$, where the elements are interpreted as row vectors; likewise, $R^{m \times 1} = R^m$, in which the elements are interpreted as column vectors. Along similar lines, if both m and n are equal to 1, then $R^{m \times n}$ reduces to the underlying entry ring, i.e., $R^{1 \times 1} = R$. Lastly, between groups of matrices with compatible dimensions, matrix multiplication is defined in accordance with its customary definition; analogously, scalar multiplication of an element from R with an element from $R^{m\times n}$ is defined as per its regular definition[1].

Third, the specifications and proofs in the upcoming sections frequently utilize "modular scaling and flooring" operators, extended coefficient-wise over polynomials and entry-wise over vectors and matrices. Specifically, suppose $p, q \in \mathbb{N}$ such that $0 < p$ and $0 < q$; moreover, let $x \in \mathbb{Z}_q$. Then, the modular scaling and flooring operator $\lfloor \cdot \rceil_{q\to p} : \mathbb{Z}_q \to \mathbb{Z}_p$ computes the following.

$$\lfloor x \rfloor_{q\to p} = \lfloor \frac{p}{q} \cdot x \rfloor \bmod p$$

Here, although not explicitly denoted, p, q, and x are lifted to the field of real numbers prior to the computation of the division and multiplication; as such, the argument to the (regular) flooring function is, conform to its definition, an element of $\mathbb{R}$. Similarly, the result of the modular reduction is lifted to $\mathbb{Z}_p$; in consequence, the final result is an element of the defined range[2]. As an example, if $p = 4$, $q = 10$, and $x = 7$, the modular scaling and flooring operator performs the computation provided below.

$$\begin{aligned}\lfloor 7 \rfloor_{10\to 4} &= \lfloor \frac{4}{10} \cdot 7 \rfloor \bmod 4 \\ &= \lfloor 0.4 \cdot 7 \rfloor \bmod 4 \\ &= \lfloor 2.8 \rfloor \bmod 4 \\ &= 2 \bmod 4 \\ &= 2\end{aligned}$$

In the context of a modular scaling and flooring operator $\lfloor \cdot \rfloor_{q\to p}$, we occasionally refer to p as the *target modulus* and q as the *source modulus*.

Regarding the discussion on Saber's PKE scheme, we exclusively consider modular scaling and flooring operators with integral power-of-two moduli, i.e., $\lfloor \cdot \rfloor_{2^{\epsilon_0}\to 2^{\epsilon_1}}$ for $\epsilon_0, \epsilon_1 \in \mathbb{N}$. As a consequence of this moduli structure, these operators possess several convenient properties. In particular, if $\epsilon_0 > \epsilon_1$, $\lfloor b \rfloor_{2^{\epsilon_0}\to 2^{\epsilon_1}}$ is equivalent to performing a right bit-shift of $\epsilon_0 - \epsilon_1$ bits on b; similarly, if $\epsilon_0 < \epsilon_1$, the corresponding operator is equivalent to carrying out a left bit-shift of $\epsilon_0 - \epsilon_1$ bits on b. More precisely, denoting the binary representation of b by $b_{\epsilon_0 - 1} \dots b_0$, the imminent derivation demonstrates the former case.

$$\begin{aligned}\lfloor b \rfloor_{2^{\epsilon_0}\to 2^{\epsilon_1}} &= \left\lfloor \frac{2^{\epsilon_1}}{2^{\epsilon_0}} \cdot b_{\epsilon_0-1} \dots b_0 \right\rfloor \bmod 2^{\epsilon_1} \\ &= \left\lfloor \frac{1}{2^{\epsilon_0-\epsilon_1}} \cdot b_{\epsilon_0-1} \dots b_0 \right\rfloor \bmod 2^{\epsilon_1} \\ &= \lfloor b_{\epsilon_0-1} \dots b_{\epsilon_0-\epsilon_1} . b_{\epsilon_0-\epsilon_1-1} \dots b_0 \rfloor \bmod 2^{\epsilon_1} \\ &= b_{\epsilon_0-1} \dots b_{\epsilon_0-\epsilon_1} \bmod 2^{\epsilon_1} \\ &= b_{\epsilon_0-1} \dots b_{\epsilon_0-\epsilon_1}\end{aligned}$$

In this derivation, considering that $\epsilon_0 > \epsilon_1$ implies $\epsilon_0 - \epsilon_1 > 0$, the third equality follows from the fact that multiplication by a factor of $\frac{1}{2^{\epsilon_0-\epsilon_1}}$ induces a left shift of the binary point by $\epsilon_0 - \epsilon_1$

[1]Actually, this implies that, in addition to an additive group, $R^{m\times n}$ is an R-module; nevertheless, for intelligibility purposes, we disregard this technicality.

[2]Technically, if $x \in \mathbb{Z}_q$ is represented by an integer in the range $[0, q-1]$, which is conventionally the case in this thesis, the (regular) modular reduction performed in $\lfloor x \rfloor_{q\to p}$ can never affect the final result of this operation; indeed, this is due to the fact that if $x \in [0, q-1]$, then $\left\lfloor \frac{p}{q} \cdot x \right\rfloor \in [0, p-1]$. Nevertheless, we still incorporate this reduction modulo p as it can serve as a convenient reminder that the eventual result of such a modular scaling and flooring operation is lifted to $\mathbb{Z}_p$; furthermore, this modular reduction makes the modular scaling and flooring operator more consistent with similar modular scaling operators, including the original operator on which it is based [15].

places. Moreover, the last equality is a consequence of a property of modular reduction with an integral power-of-two modulus. Namely, for any modulus 2^ϵ with $\epsilon \in \mathbb{N}$, modular reduction is equivalent to setting all bits *more* significant than the ϵ-th bit to 0; in turn, this is tantamount to exclusively evaluating the ϵ *least* significant bits. Therefore, since $b_{\epsilon_0-1} \dots b_{\epsilon_0-\epsilon_1}$ consists of $\epsilon_0 - (\epsilon_0 - \epsilon_1) = \epsilon_1$ bits, the modular reduction with modulus 2^{ϵ_1} effectively reduces to the identity function.

Analogously, the following derivation shows that if $\epsilon_0 < \epsilon_1$, $\lfloor b \rfloor_{2^{\epsilon_0} \to 2^{\epsilon_1}}$ practically performs a left bit-shift of $\epsilon_1 - \epsilon_0$ bits on b.

$$\begin{aligned}
\lfloor b \rfloor_{2^{\epsilon_0} \to 2^{\epsilon_1}} &= \left\lfloor \frac{2^{\epsilon_1}}{2^{\epsilon_0}} \cdot b_{\epsilon_0-1} \dots b_0 \right\rfloor \bmod 2^{\epsilon_1} \\
&= \left\lfloor 2^{\epsilon_1-\epsilon_0} \cdot b_{\epsilon_0-1} \dots b_0 \right\rfloor \bmod 2^{\epsilon_1} \\
&= \left\lfloor b_{\epsilon_0-1} \dots b_0 0^{\epsilon_1-\epsilon_0} \right\rfloor \bmod 2^{\epsilon_1} \\
&= b_{\epsilon_0-1} \dots b_0 0^{\epsilon_1-\epsilon_0} \bmod 2^{\epsilon_1} \\
&= b_{\epsilon_0-1} \dots b_0 0^{\epsilon_1-\epsilon_0}
\end{aligned}$$

Here, because $\epsilon_1 - \epsilon_0 > 0$, the third equality results from the fact that multiplication by a factor of $2^{\epsilon_1-\epsilon_0}$ induces a right shift of the binary point by $\epsilon_1 - \epsilon_0$ places; consequently, this multiplication precisely introduces $\epsilon_1 - \epsilon_0$ least significant zero bits. Additionally, since $b_{\epsilon_0-1} \dots b_0 0^{\epsilon_1-\epsilon_0}$ comprises exactly $\epsilon_0 + (\epsilon_1 - \epsilon_0) = \epsilon_1$ bits, the final equality holds due to the aforementioned property of modular reduction with a power-of-two modulus.

Associated with each modular scaling and flooring operator, we straightforwardly define coefficient-wise and entry-wise extensions for, respectively, polynomials and vectors/matrices. Specifically, the coefficient-wise extension of such an operator independently applies the operator to each coefficient of the considered polynomial; naturally, the coefficients of this polynomial must be elements from the operator's domain. Similarly, the entry-wise extension of such an operator individually applies the operator to each entry of the vector/matrix in question; analogous to the coefficients of a polynomial, the entries of this vector/matrix must be compatible with the operator. Furthermore, these extensions may be combined; in particular, a coefficient-wise extension of an operator can have an entry-wise extension. As such, we also obtain modular scaling and flooring operators applicable to polynomial vectors. A necessary consequence of the straightforward definitions of these extensions is that, to determine the outcome of the application of an extended operator, it suffices to compute the outcomes of the application of the underlying operator on each coefficient or entry of, respectively, the considered polynomial or vector/matrix. Lastly, by abuse of notation, these extensions are denoted identically to their underlying operators.

In addition to the above-introduced modular scaling and flooring operator, the remainder occasionally refers to "modular scaling and rounding" operators of the form $\lfloor \cdot \rceil_{q \to p} : \mathbb{Z}_q \to \mathbb{Z}_p$, where $p, q \in \mathbb{N}$ such that $0 < p$ and $0 < q$. For a certain p and q, this operator is identical to its modular scaling and flooring counterpart, except that it uses rounding instead of flooring; moreover, this operator is extended analogously to the modular scaling and flooring operator.

As a last preliminary related to operators, we cover the overloading and extending of the modulo operator. Specifically, for any $p, q \in \mathbb{N}$ such that $0 < p$, $0 < q$, and $p \mid q$, we let "mod p" denote a well-defined mapping from $\mathbb{Z}_q$ to $\mathbb{Z}_p$. In terms of their computations, these mappings are equivalent to the conventional modulo operator when lifting their operands and results to $\mathbb{Z}$ and $\mathbb{Z}_p$, respectively. Furthermore, these mappings are extended similarly to the modular reduction and flooring/rounding operators.

Fourth, we introduce coefficient-wise and entry-wise extensions of sampling; indeed, these extensions are analogous to the extensions defined for the operators mentioned above. More precisely, while regular sampling involves sampling a single numerical value from the considered probability distribution, extension sampling, coefficient-wise or entry-wise, entails repeatedly and independently sampling numerical values until the required number of values is obtained. For instance, suppose we desire to sample a vector of polynomials from $\beta_\mu(R_q^{m \times 1})$, where $0 < m$. Then, since the desired element is an m-dimensional vector with entries from R_q, the sampling's entry-wise extension independently samples m elements from $\beta_\mu(R_q)$. In turn, because R_q consists of polynomial elements, the sampling's coefficient-wise extension independently samples all necessary coefficients from $\beta_\mu(\mathbb{Z}_q)$. At last, these coefficients are obtained by (regularly) sampling from the centered binomial distribution with parameter μ and lifting the sampled values to $\mathbb{Z}_q$.

Fifth, the Saber cipher suite defines a set of configuration parameters that determines the quotient ring structure, moduli, distributions, vectors, and matrices utilized in the schemes [14]. Specifically, this set comprises the exponent ϵ_n corresponding to the degree $n = 2^{\epsilon_n}$ of the polynomial modulus $X^n + 1$; the exponents ϵ_t, ϵ_p, and ϵ_q associated with the moduli $t = 2^{\epsilon_t}$, $p = 2^{\epsilon_p}$, and $q = 2^{\epsilon_q}$, respectively; the centered binomial distribution's parameter μ; and the dimension l for the vectors and matrices[3]. The chosen parameter values influence multiple essential properties of Saber's schemes; particularly, these values impact the security level, failure probability, and required bandwidth and storage. As such, assigning values to these parameters is a critical and delicate endeavor. Nevertheless, technically, the parameters must merely satisfy the following list of requirements for the schemes to conform to their specifications.

1. **Moduli Exponent Ordering**
   The exponents of the moduli must adhere to $0 < \epsilon_t + 1 < \epsilon_p < \epsilon_q$ or, equivalently, $1 \leq \epsilon_t + 1 \ \wedge \ \epsilon_t + 2 \leq \epsilon_p \ \wedge \ \epsilon_p + 1 \leq \epsilon_q$. Consequently, $0 < 2 \cdot t < p < q$ and $2 \cdot t \mid p \mid q$.

2. **Valid Polynomial Degree**
   The exponent ϵ_n corresponding to the degree n of the polynomial modulus $X^n + 1$ must be a natural number, i.e., $\epsilon_n \in \mathbb{N}$.

3. **Valid Vector/Matrix Dimensions**
   The dimension of the employed vectors and matrices must be a strictly positive natural number; that is, $l \in \mathbb{N}$ and $0 < l$.

4. **Small Centered Binomial Distribution Parameter**
   The parameter μ of the utilized centered binomial distribution must be strictly less than p, i.e., $\mu < p$.

Finally, Saber uses several predefined constants in its schemes [14]. In particular, these constants are utilized with certain modular scaling and flooring operators to mimic modular scaling and rounding operators, retaining the schemes' desired security and correctness properties. The rationale for employing modular scaling and flooring operators over their rounding counterparts is that the former facilitate the implementation process; specifically, this is because most programming languages default to effectively flooring the result of computations with integers. Concretely, the constants are denoted by $h_1, h_2 \in R_q$, and $\mathbf{h} \in R_q^{l \times 1}$; moreover, the former two constants are defined as $h_1 = \sum_{i=0}^{n-1} \frac{q}{2 \cdot p} \cdot X^i = \sum_{i=0}^{n-1} 2^{\epsilon_q - \epsilon_p - 1} \cdot X^i$ and $h_2 = \sum_{i=0}^{n-1} (\frac{p}{4} - \frac{p}{4 \cdot t}) \cdot X^i = \sum_{i=0}^{n-1} (2^{\epsilon_p - 2} - 2^{\epsilon_p - \epsilon_t - 2}) \cdot X^i$, while $\mathbf{h}$ is defined as the vector with all entries equal to h_1. Here, the q, p, t, and n are the aforementioned moduli and degree defined by the ϵ_q, ϵ_p, ϵ_t, and ϵ_n parameters, respectively.

Elaborating on the above, we demonstrate the manner in which Saber's schemes utilize particular modular scaling and flooring operators in combination with the constants to mimic modular scaling and rounding operators. Predominantly, considering an arbitrary element $\mathbf{v} \in R_q^{l \times 1}$, the schemes

[3]The matrices in Saber are all square, so a single parameter suffices to specify their dimension.

mimic $\lfloor \mathbf{v} \rceil_{q \to p}$ by means of $\lfloor \mathbf{v} + \mathbf{h} \rfloor_{q \to p}$; indeed, $\lfloor \mathbf{v} \rceil_{q \to p} = \lfloor \mathbf{v} + \mathbf{h} \rfloor_{q \to p}$. To see why this holds, let $x_{\epsilon_q - 1} \dots x_{\epsilon_q - \epsilon_p} x_{\epsilon_q - \epsilon_p - 1} x_{\epsilon_q - \epsilon_p - 2} \dots x_0$ be the binary representation of an arbitrary element $x \in \mathbb{Z}_q$; then, we can distinguish two cases: $x_{\epsilon_q - \epsilon_p - 1} = 0$ and $x_{\epsilon_q - \epsilon_p - 1} = 1$. In the former case, the addition of $2^{\epsilon_q - \epsilon_p - 1}$ only affects the value of $x_{\epsilon_q - \epsilon_p - 1}$, changing it to 1; in particular, the remainder of x's bits are unaffected by this addition since it induces no carries. The derivation below shows the equality between $\lfloor x + 2^{\epsilon_q - \epsilon_p - 1} \rfloor_{q \to p}$ and $\lfloor x \rceil_{q \to p}$ in this case.

$$\begin{aligned}
\lfloor x + 2^{\epsilon_q - \epsilon_p - 1} \rfloor_{q \to p} &= \lfloor x_{\epsilon_q - 1} \dots x_{\epsilon_q - \epsilon_p} 0 x_{\epsilon_q - \epsilon_p - 2} \dots x_0 + 0^{\epsilon_p} 1 0^{\epsilon_q - \epsilon_p - 1} \rfloor_{q \to p} \\
&= \lfloor x_{\epsilon_q - 1} \dots x_{\epsilon_q - \epsilon_p} 1 x_{\epsilon_q - \epsilon_p - 2} \dots x_0 \rfloor_{q \to p} \\
&= \lfloor x_{\epsilon_q - 1} \dots x_{\epsilon_q - \epsilon_p} . 1 x_{\epsilon_q - \epsilon_p - 2} \dots x_0 \rfloor \\
&= x_{\epsilon_q - 1} \dots x_{\epsilon_q - \epsilon_p} \\
&= \lfloor x_{\epsilon_q - 1} \dots x_{\epsilon_q - \epsilon_p} . 0 x_{\epsilon_q - \epsilon_p - 2} \dots x_0 \rceil \\
&= \lfloor x_{\epsilon_q - 1} \dots x_{\epsilon_q - \epsilon_p} 0 x_{\epsilon_q - \epsilon_p - 2} \dots x_0 \rceil_{q \to p} \\
&= \lfloor x \rceil_{q \to p}
\end{aligned}$$

In the other case, i.e., when $x_{\epsilon_q - \epsilon_p - 1} = 1$, the addition of $2^{\epsilon_q - \epsilon_p - 1}$ changes the value of $x_{\epsilon_q - \epsilon_p - 1}$ to 0 and produces a carry. Naturally, this carry is added to bit $x_{\epsilon_q - \epsilon_p}$; in turn, the addition of this carry might affect the value of the ϵ_p *most* significant bits of x. Nevertheless, also in this case, $\lfloor x + 2^{\epsilon_q - \epsilon_p - 1} \rfloor_{q \to p}$ is equal to $\lfloor x \rceil_{q \to p}$. Namely, as can be extracted from the preceding derivation, $x_{\epsilon_q - \epsilon_p - 1}$ is the first bit that follows the binary point after evaluating the multiplication of x with $\frac{p}{q}$. In consequence, since $x_{\epsilon_q - \epsilon_p - 1} = 1$, $\lfloor x \rceil_{q \to p}$ effectively drops the $\epsilon_q - \epsilon_p$ *least* significant bits of x and adds 1 to the remaining value. Certainly, because this remaining value equals $x_{\epsilon_q - 1} \dots x_{\epsilon_q - \epsilon_p}$, this addition is equivalent to the aforementioned carry addition that occurs when adding $2^{\epsilon_q - \epsilon_p - 1}$ to the initial value of x. Thus, albeit in a different order, $\lfloor x + 2^{\epsilon_q - \epsilon_p - 1} \rfloor_{q \to p}$ and $\lfloor x \rceil_{q \to p}$ both essentially drop the $\epsilon_q - \epsilon_p$ *least* significant bits of x and add 1 to $x_{\epsilon_q - \epsilon_p}$; as such, even if $x_{\epsilon_q - \epsilon_p - 1} = 1$, $\lfloor x + 2^{\epsilon_q - \epsilon_p - 1} \rfloor_{q \to p}$ equals $\lfloor x \rceil_{q \to p}$.

From the above-derived equality, we can directly infer the analog equalities between the extensions of the considered operators; particularly, this direct inference is possible due to the straightforward definitions of the extensions, as explicated above. Naturally, regarding the extensions of the modular scaling and flooring operator, the polynomial or vector/matrix that is added to their operand must, respectively, solely comprise coefficients or entries equal to $2^{\epsilon_q - \epsilon_p - 1}$. An analogous argument holds for the combination of these extensions. Consequently, since each entry of $\mathbf{h}$ is equal to h_1 and, in turn, each coefficient of h_1 is equal to $2^{\epsilon_q - \epsilon_p - 1}$, we can derive the following equality for all $\mathbf{v} \in R_q^{l \times 1}$.

$$\lfloor \mathbf{v} + \mathbf{h} \rfloor_{q \to p} = \lfloor \mathbf{v} \rceil_{q \to p}$$

3.2 Public-Key Encryption Scheme

As all PKE schemes, Saber's PKE scheme comprises a triple of algorithms: a key generation algorithm, an encryption algorithm, and a decryption algorithm. Throughout, these algorithms are referred to as Saber.KeyGen, Saber.Enc, and Saber.Dec, respectively; moreover, Saber's PKE scheme is denoted by Saber.PKE. As such, we have Saber.PKE = (Saber.KeyGen, Saber.Enc, Saber.Dec).

This section extensively discusses and analyzes Saber.PKE. Specifically, first, Section 3.2.1 discusses the specifications of Saber.PKE's algorithms; subsequently, Section 3.2.2 and Section 3.2.3 respectively analyze the scheme's security and correctness.

3.2.1 Specification

Proceeding in the logical order of execution, we present and elaborate on Saber.PKE's algorithms; that is, we cover Saber.KeyGen first, followed by Saber.Enc and, at last, Saber.Dec. For all of these algorithms, we adopt the specifications from the original paper [14].

Foremost, several of Saber.PKE's algorithms are defined with respect to a publicly known and efficiently evaluable algorithm gen. Given an input seed, i.e., a bit-string of a certain length, gen generates an element of $R_q^{l \times l}$, where q and l respectively denote the modulus and dimension introduced in Section 3.1. Furthermore, it is imperative that, on the same input, this algorithm invariably produces the same output. In other words, gen must be a well-defined mathematical function.

Addressing the first algorithm's specification, Algorithm 1 contains the specification of Saber.KeyGen. This algorithm takes no inputs and returns a key pair as output; specifically, this key pair is produced as follows. Initially, Saber.KeyGen samples $\text{seed}_{\mathbf{A}}$, a bit-string of length 256, uniformly at random. Subsequently, the algorithm calls gen with input $\text{seed}_{\mathbf{A}}$ and assigns the result, an element of $R_q^{l \times l}$, to $\mathbf{A}$; i.e., matrix $\mathbf{A}$ is obtained via a call to $\mathsf{gen}(\text{seed}_{\mathbf{A}})$, giving $\mathbf{A} \in R_q^{l \times l}$. Then, the algorithm acquires the secret key $\mathbf{s}$ by sampling from $\beta_\mu(R_q^{l \times 1})$. Thereafter, Saber.KeyGen computes the second and last part of the public key, i.e., $\mathbf{b}$, by effectively modular scaling and rounding the matrix-vector multiplication of $\mathbf{A}$ and $\mathbf{s}$ from modulo q to modulo p. Lastly, the algorithm returns the generated public key pk, which constitutes the tuple $(\text{seed}_{\mathbf{A}}, \mathbf{b})$, and secret key sk, which equals $\mathbf{s}$.

Algorithm 1 Saber's Key Generation Algorithm

```
1: procedure Saber.KeyGen()
2:     $\text{seed}_{\mathbf{A}} \leftarrow\$ \mathcal{U}(\{0,1\}^{256})$
3:     $\mathbf{A} \leftarrow \mathsf{gen}(\text{seed}_{\mathbf{A}})$
4:     $\mathbf{s} \leftarrow\$ \beta_\mu(R_q^{l \times 1})$
5:     $\mathbf{b} \leftarrow \lfloor \mathbf{A} \cdot \mathbf{s} + \mathbf{h} \rceil_{q \to p}$
6:     return $\mathsf{pk} := (\text{seed}_{\mathbf{A}}, \mathbf{b}),\ \mathsf{sk} := \mathbf{s}$
7: end procedure
```

Next, Algorithm 2 provides the specification of Saber.Enc. In contrast to Saber.KeyGen, this algorithm does take inputs; more precisely, it takes a public key pk, which is a tuple generated in accordance with Saber.KeyGen, and a message m, which is encoded as an element of R_2. With regards to the latter, the message space $\mathcal{M}$ of Saber.PKE is the set of bit-strings of length n, where n is the parameter-induced value mentioned in Section 3.1; that is, $\mathcal{M} = \{0,1\}^n$. This conveys that both Saber.Enc and Saber.Dec are only capable of, respectively, encrypting and decrypting messages representable by bit-strings of length n. To use such messages in the algorithms' computations, they are, as stated above, encoded as elements of R_2. In particular, considering any message $m_{plain} = m_0 m_1 \ldots m_{n-2} m_{n-1}$ such that $\forall_{0 \leq i < n} : m_i \in \{0,1\}$, the corresponding encoded element equals $m_{encoded} = \sum_{i=0}^{n-1} m_i \cdot X^i$. Decoding an encoded message is trivially accomplished by performing the inverse operation, i.e., through concatenation of the encoded message's coefficients. Provided with the input, Saber.Enc starts by generating matrix $\mathbf{A}$ and sampling secret vector $\mathbf{s}'$ in an identical manner to, respectively, the generation of $\mathbf{A}$ and sampling of $\mathbf{s}$ in Saber.KeyGen. Furthermore, Saber.Enc's subsequent computation of $\mathbf{b}'$, which constitutes one of two parts of the eventual ciphertext, is comparable to Saber.KeyGen's computation of $\mathbf{b}$; namely, Saber.Enc computes $\mathbf{b}'$ by effectively modular scaling and rounding the result of $\mathbf{A}^T \cdot \mathbf{s}'$ from modulo q to modulo p. Afterward, Saber.Enc proceeds by constructing the ciphertext's remaining part. To this end, the algorithm first computes an intermediate value v' from $\mathbf{b}$, $\mathbf{s}'$, and h_1. In this computation, $\mathbf{s}'$ and h_1 are reduced modulo p which, as discussed in the previous section, entails reducing all coefficients (of each entry) modulo p. Indeed, since each coefficient is an element of $\mathbb{Z}_q$ and $p \mid q$, these modular reductions are well-defined; moreover, because the resulting coefficients are elements of $\mathbb{Z}_p$, $\mathbf{s}' \bmod p$ and $h_1 \bmod p$ are elements of $R_p^{l \times 1}$ and R_p, respectively. Subsequent to the computation of v', the algorithm obtains the remaining part of the ciphertext, i.e., c_m, by adding a scaled version of the encoded input message to v' and, thereafter, modular scaling and flooring the result from modulo p to modulo $2 \cdot t$. As such, intuitively, (the most significant bits of) the coefficients of v' collectively serve as the key employed to encrypt the message m. At last, Saber.Enc returns the ciphertext c, which is comprised of the tuple $(c_m, \mathbf{b}')$.

Algorithm 2 Saber's Encryption Algorithm

```
1: procedure Saber.Enc(pk := (seed_A, b), m)
2:     $\mathbf{A} \leftarrow \mathsf{gen}(\mathrm{seed}_{\mathbf{A}})$
3:     $\mathbf{s}' \leftarrow_{\$} \beta_\mu(R_q^{l \times 1})$
4:     $\mathbf{b}' \leftarrow \lfloor \mathbf{A}^T \cdot \mathbf{s}' + \mathbf{h} \rfloor_{q \to p}$
5:     $v' \leftarrow \mathbf{b}^T \cdot (\mathbf{s}' \bmod p) + (h_1 \bmod p)$
6:     $c_m \leftarrow \lfloor v' + \lfloor m \rfloor_{2 \to p} \rfloor_{p \to 2 \cdot t}$
7:     return $c := (c_m, \mathbf{b}')$
8: end procedure
```

Finally, Algorithm 3 presents the specification of Saber.Dec. As input, this algorithm takes a secret key sk, generated in accordance with Saber.KeyGen, and a ciphertext c, produced as per Saber.Enc. Following, Saber.Dec computes v identically to the manner in which Saber.Enc computes v'; however, instead of $\mathbf{b}$ and $\mathbf{s}'$, Saber.Dec uses $\mathbf{b}'$ and $\mathbf{s}$. Subsequently, Saber.Dec attempts to extract the encoded message from the given c_m by subtracting a scaled version of this value from v, adding $h_2 \bmod p$, and, ultimately, modular scaling and flooring the result from modulo p to modulo 2. Lastly, Saber.Dec returns the result of this extraction attempt, i.e., m'.

Algorithm 3 Saber's Decryption Algorithm

```
1: procedure Saber.Dec(sk := $\mathbf{s}$, $c := (c_m, \mathbf{b}')$)
2:     $v \leftarrow \mathbf{b}'^T \cdot (\mathbf{s} \bmod p) + (h_1 \bmod p)$
3:     $m' \leftarrow \lfloor v - \lfloor c_m \rfloor_{2 \cdot t \to p} + (h_2 \bmod p) \rfloor_{p \to 2}$
4:     return $m'$
5: end procedure
```

A comprehensive and minute analysis of Saber.PKE's correctness, i.e., the probability that Saber.Dec unerringly extracts a message from the corresponding encryption produced by Saber.Enc[4], is carried out in Section 3.2.3; nevertheless, we presently provide a concise, more cursory explication of this property to demonstrate the relevant aspects and concepts. Foremost, recall that due to Saber's parameter requirements specified in Section 3.1, we have $q > p > 2 \cdot t$. Consequently, the $\lfloor \cdot \rceil_{q \to p}$ and $\lfloor \cdot \rfloor_{p \to 2 \cdot t}$ operators scale their operands to a smaller modulus, thereby introducing an error; moreover, the $\lfloor \cdot \rfloor_{2 \cdot t \to p}$ operator effectively reduces to (real) multiplication of its operand with $\frac{p}{2 \cdot t}$. Furthermore, as derived previously, for all $\mathbf{v} \in R_q^{l \times 1}$, we have $\lfloor \mathbf{v} + \mathbf{h} \rfloor_{q \to p} = \lfloor \mathbf{v} \rceil_{q \to p}$; particularly, this holds for $\mathbf{v} = \mathbf{A} \cdot \mathbf{s}$ and $\mathbf{v} = \mathbf{A}^T \cdot \mathbf{s}'$, where $\mathbf{A}$, $\mathbf{s}$, and $\mathbf{s}'$ denote the artifacts employed in Saber.KeyGen and Saber.Enc. As such, we can represent the $\mathbf{b}$, $\mathbf{b}'$, and c_m computed in, respectively, Saber.KeyGen, Saber.Enc, and Saber.Enc as follows. Here, $\mathbf{e_b}$, $\mathbf{e_{b'}}$, and e_{c_m} represent the errors introduced by the modular scaling and flooring operators.

$$\mathbf{b} = \lfloor \mathbf{A} \cdot \mathbf{s} + \mathbf{h} \rfloor_{q \to p} = \lfloor \mathbf{A} \cdot \mathbf{s} \rceil_{q \to p} = \frac{p}{q} \cdot (\mathbf{A} \cdot \mathbf{s}) + \mathbf{e_b}$$

$$\mathbf{b}' = \lfloor \mathbf{A}^T \cdot \mathbf{s}' + \mathbf{h} \rfloor_{q \to p} = \lfloor \mathbf{A}^T \cdot \mathbf{s}' \rceil_{q \to p} = \frac{p}{q} \cdot (\mathbf{A}^T \cdot \mathbf{s}') + \mathbf{e_{b'}}$$

$$c_m = \lfloor v' + \lfloor m \rfloor_{2 \to p} \rfloor_{p \to 2 \cdot t} = \frac{2 \cdot t}{p} \cdot (v' + \lfloor m \rfloor_{2 \to p}) + e_{c_m}$$

In the above, for reasons of intelligibility, we slightly abuse notation by not explicitly denoting the lifting of elements. In fact, this is additionally the case for the remaining derivations in this correctness-related explication.

[4]Naturally, this is merely an intuitive description of Saber.PKE's correctness; for the corresponding formal definition(s), refer to Section 3.2.3.

Employing the above-inferred expression for c_m, we deduce an alternative expression for $\lfloor c_m \rceil_{2\cdot t \to p}$; indeed, this is one of the terms utilized in Saber.Dec to compute m'.

$$\begin{aligned}
\lfloor c_m \rceil_{2\cdot t\to p} &= \lfloor \frac{2\cdot t}{p} \cdot (v' + \lfloor m \rceil_{2\to p}) + e_{c_m} \rceil_{2\cdot t \to p} \\
&= \frac{p}{2\cdot t} \cdot (\frac{2\cdot t}{p} \cdot (v' + \lfloor m \rceil_{2\to p}) + e_{c_m}) \\
&= v' + \lfloor m \rceil_{2\to p} + \frac{p}{2\cdot t} \cdot e_{c_m}
\end{aligned}$$

Penultimately, as a useful intermediate step preceding the final derivation, we rewrite the expression $v - v'$, where v and v' are the artifacts used in Saber.Dec and Saber.Enc. In this endeavor, we utilize the previously derived expressions for $\mathbf{b}$ and $\mathbf{b}'$.

$$\begin{aligned}
v - v' &= (\mathbf{b}'^T \cdot (\mathbf{s} \bmod p) + (h_1 \bmod p)) - (\mathbf{b}^T \cdot (\mathbf{s}' \bmod p) + (h_1 \bmod p)) \\
&= \mathbf{b}'^T \cdot (\mathbf{s} \bmod p) - \mathbf{b}^T \cdot (\mathbf{s}' \bmod p) \\
&= (\frac{p}{q} \cdot (\mathbf{A}^T \cdot \mathbf{s}') + \mathbf{e}_{\mathbf{b}'})^T \cdot (\mathbf{s} \bmod p) - (\frac{p}{q} \cdot (\mathbf{A} \cdot \mathbf{s}) + \mathbf{e}_{\mathbf{b}})^T \cdot (\mathbf{s}' \bmod p) \\
&= (\frac{p}{q} \cdot (\mathbf{s}'^T \cdot \mathbf{A}) + \mathbf{e}_{\mathbf{b}'}^T) \cdot (\mathbf{s} \bmod p) - (\frac{p}{q} \cdot (\mathbf{s}^T \cdot \mathbf{A}^T) + \mathbf{e}_{\mathbf{b}}^T) \cdot (\mathbf{s}' \bmod p) \\
&= \frac{p}{q} \cdot (\mathbf{s}'^T \cdot \mathbf{A} \cdot (\mathbf{s} \bmod p)) + \mathbf{e}_{\mathbf{b}'}^T \cdot (\mathbf{s} \bmod p) - (\frac{p}{q} \cdot (\mathbf{s}^T \cdot \mathbf{A}^T \cdot (\mathbf{s}' \bmod p)) + \mathbf{e}_{\mathbf{b}}^T \cdot (\mathbf{s}' \bmod p)) \\
&= \mathbf{e}_{\mathbf{b}'}^T \cdot (\mathbf{s} \bmod p) - \mathbf{e}_{\mathbf{b}}^T \cdot (\mathbf{s}' \bmod p)
\end{aligned}$$

Here, the initial five equalities follow from basic operator properties or trivial substitutions and simplifications. Specifically, in order, these equalities are induced by the substitutions of v and v', the simplification that eliminates $h_1 \bmod p$, the substitutions of $\mathbf{b}'$ and $\mathbf{b}$, the distributivity of the transpose over matrix multiplication, and the distributivity of matrix multiplication over matrix addition. Somewhat less evident, the last equality is a consequence of the fact that $\frac{p}{q} \cdot (\mathbf{s}'^T \cdot \mathbf{A} \cdot (\mathbf{s} \bmod p)) = \frac{p}{q} \cdot (\mathbf{s}^T \cdot \mathbf{A}^T \cdot (\mathbf{s}' \bmod p))$. In turn, this is implied by $\mathbf{s}'^T \cdot \mathbf{A} \cdot (\mathbf{s} \bmod p) = (\mathbf{s}^T \cdot \mathbf{A}^T \cdot (\mathbf{s}' \bmod p))^T = \mathbf{s}^T \cdot \mathbf{A}^T \cdot (\mathbf{s}' \bmod p)$; certainly, since $\mathbf{s}^T \cdot \mathbf{A}^T \cdot (\mathbf{s}' \bmod p)$ evaluates to a single value, i.e., not a matrix (nor a vector), the transpose effectively reduces to the identity function.

Lastly, leveraging the expressions deduced for $\lfloor c_m \rceil_{2\cdot t\to p}$ and $v - v'$, we rewrite Saber.Dec's computation of m' in the following manner.

$$\begin{aligned}
m' &= \lfloor v - \lfloor c_m \rceil_{2\cdot t\to p} + (h_2 \bmod p) \rceil_{p\to 2} \\
&= \lfloor v - (v' + \lfloor m \rceil_{2\to p} + \frac{p}{2\cdot t} \cdot e_{c_m}) + (h_2 \bmod p) \rceil_{p\to 2} \\
&= \lfloor (v - v') - \lfloor m \rceil_{2\to p} - \frac{p}{2\cdot t} \cdot e_{c_m} + (h_2 \bmod p) \rceil_{p\to 2} \\
&= \lfloor \mathbf{e}_{\mathbf{b}'}^T \cdot (\mathbf{s} \bmod p) - \mathbf{e}_{\mathbf{b}}^T \cdot (\mathbf{s}' \bmod p) - \lfloor m \rceil_{2\to p} - \frac{p}{2\cdot t} \cdot e_{c_m} + (h_2 \bmod p) \rceil_{p\to 2} \\
&= \lfloor \lfloor m \rceil_{2\to p} + \mathbf{e}_{\mathbf{b}'}^T \cdot (\mathbf{s} \bmod p) - \mathbf{e}_{\mathbf{b}}^T \cdot (\mathbf{s}' \bmod p) - \frac{p}{2\cdot t} \cdot e_{c_m} + (h_2 \bmod p) \rceil_{p\to 2} \\
&= m + \lfloor \mathbf{e}_{\mathbf{b}'}^T \cdot (\mathbf{s} \bmod p) - \mathbf{e}_{\mathbf{b}}^T \cdot (\mathbf{s}' \bmod p) - \frac{p}{2\cdot t} \cdot e_{c_m} + (h_2 \bmod p) \rceil_{p\to 2}
\end{aligned}$$

Akin to the preceding derivation's initial equalities, the first four equalities in this derivation are induced by trivial substitutions and fundamental operator properties; contrarily, the legitimacy of the latter two equalities is more difficult to discern. Namely, the penultimate equality is implied by $-\lfloor m \rceil_{p\to 2} = \lfloor m \rceil_{p\to 2}$; furthermore, the final equality follows from the fact that for all $x \in R_p$, $\lfloor \lfloor m \rceil_{2\to p} + x \rceil_{p\to 2} = m + \lfloor x \rceil_{p\to 2}$. For reasons of conciseness and relevance, the validity of these properties is not corroborated here; however, trivially extended to these properties, proofs for analogous properties are provided by the correctness analysis in Section 3.2.3. Considering the final expression, we see that $m' = m$ if and only if $\lfloor \mathbf{e}_{\mathbf{b}'}^T \cdot (\mathbf{s} \bmod p) - \mathbf{e}_{\mathbf{b}}^T \cdot (\mathbf{s}' \bmod p) - \frac{p}{2\cdot t} \cdot e_{c_m} + (h_2 \bmod p) \rfloor_{p\to 2} = 0$. In turn, this latter equality holds if and only if each coefficient of $\mathbf{e}_{\mathbf{b}'}^T \cdot (\mathbf{s} \bmod p) - \mathbf{e}_{\mathbf{b}}^T \cdot (\mathbf{s}' \bmod p) - \frac{p}{2\cdot t} \cdot e_{c_m} + (h_2 \bmod p)$ lies in the discrete range $[0, 2^{\epsilon_p - 1})$, i.e., each of these coefficients has its ϵ_p-th bit set to 0; certainly, this is due to the fact that the considered $\lfloor \cdot \rfloor_{p\to 2}$ operator effectively performs a right bit-shift of $\epsilon_p - 1$ bits on each coefficient (which has a total of ϵ_p bits). Concluding, since the expression $\mathbf{e}_{\mathbf{b}'}^T \cdot (\mathbf{s} \bmod p) - \mathbf{e}_{\mathbf{b}}^T \cdot (\mathbf{s}' \bmod p) - \frac{p}{2\cdot t} \cdot e_{c_m} + (h_2 \bmod p)$ is almost entirely contingent on the randomly sampled artifacts from Saber.PKE's algorithms, there exists a positive probability that at least one of its coefficients equals 1 and, hence, $m' \neq m$; surely, this indicates that Saber.PKE is not perfectly correct.

3.2.2 Security

As alluded to in Section 1.1, assuming the hardness of the MLWR problem, Saber.PKE is supposed to be IND-CPA secure. In the ensuing, we carry out a manual analysis of Saber.PKE with respect to this property; specifically, we devise a corresponding code-based, game-playing security proof. This manual analysis differs from the presently established analyses of Saber.PKE's security in the particular scheme that the code-based, game-playing security proof considers. Namely, the presently established analyses construct such a proof for the IND-RND security of Saber's KE scheme; subsequently, they essentially infer Saber.PKE's IND-CPA security from this security property of the KE scheme [14,33]. In contrast, we create a code-based, game-playing security proof directly for Saber.PKE; this approach facilitates the corresponding formal verification endeavor in EasyCrypt.

Concretely, the forthcoming manual analysis proceeds as follows. First, we define the relevant security property and hardness assumptions as (code-based) security games. Second, in terms of these security games, we formalize the security claim. Lastly, we prove this security claim by means of a game-playing proof.

Security Property and Hardness Assumptions

As aforementioned, in a code-based, game-playing cryptographic proof, hardness assumptions and security properties are formalized through games, i.e., probabilistic programs defined with respect to a relevant adversary. Regarding the IND-CPA security property for PKE schemes, this game straightforwardly formalizes the corresponding scenario explicated in Section 2.2. More precisely, considering any relevant adversary $\mathcal{A} = (\mathsf{P}, \mathsf{D})$ and PKE = (KeyGen, Enc, Dec), the general variant of this game is defined in Figure 3.1.

```
$\mathrm{Game}_{\mathcal{A},\mathrm{PKE}}^{\text{IND-CPA}}$
$u \leftarrow\$ \mathcal{U}(\{0,1\})$
$(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}()$
$(m_0, m_1) \leftarrow \mathcal{A}.\mathsf{P}(\mathsf{pk})$
$c \leftarrow \mathsf{Enc}(\mathsf{pk}, m_u)$
$u' \leftarrow \mathcal{A}.\mathsf{D}(\mathsf{pk}, c)$
return $(u' = u)$
```

Figure 3.1: The General IND-CPA Game

The advantage of adversary $\mathcal{A} = (\mathsf{P}, \mathsf{D})$ against $\mathrm{Game}_{\mathcal{A},\mathrm{PKE}}^{\text{IND-CPA}}$ denotes $\mathcal{A}$'s nontrivial probability of winning $\mathrm{Game}_{\mathcal{A},\mathrm{PKE}}^{\text{IND-CPA}}$; indeed, this captures the extent to which $\mathcal{A}$ is capable of extracting information about a message from its encryption. Considering the trivially achievable probability of winning equals $\frac{1}{2}$, the advantage of $\mathcal{A}$ against $\mathrm{Game}^{\text{IND-CPA}}_{\mathcal{A},\text{PKE}}$ is defined as follows.

$$\mathsf{Adv}^{\text{IND-CPA}}_{\text{PKE}}(\mathcal{A}) = \left|\Pr\left[\mathrm{Game}^{\text{IND-CPA}}_{\mathcal{A},\text{PKE}} = 1\right] - \frac{1}{2}\right|$$

Adhering to (the stipulations of) the scenario described in Section 2.2, $\mathrm{Game}^{\text{IND-CPA}}_{\mathcal{A},\text{PKE}}$ does not impose any restrictions on the considered adversary, except that it should be a relevant adversary. In this manner, $\mathrm{Game}^{\text{IND-CPA}}_{\mathcal{A},\text{PKE}}$ implicitly conveys that the adversary may arbitrarily compute and store data during the game's execution[5]; particularly, in the call to $\mathcal{A}.\mathsf{D}(\mathsf{pk}, c)$, the adversary may still have knowledge of m_0, m_1, and all computations performed in $\mathcal{A}.\mathsf{P}(\mathsf{pk})$. In other words, the adversary is capable of maintaining a *state*. Furthermore, $\mathrm{Game}^{\text{IND-CPA}}_{\mathcal{A},\text{PKE}}$ allows the adversary to pick the messages itself, which precludes the advantage from being dependent on the particular construction of the messages. As such, if $\mathsf{Adv}^{\text{IND-CPA}}_{\text{PKE}}(\mathcal{A})$ is negligible for all relevant adversaries, PKE indeed is IND-CPA secure.

Concretizing the general IND-CPA game for Saber.PKE, we obtain the game depicted in Figure 3.2.

```
$\mathrm{Game}^{\text{IND-CPA}}_{\mathcal{A},\text{Saber.PKE}}$
1 : $u \leftarrow\$ \mathcal{U}(\{0,1\})$
2 : $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Saber.KeyGen}()$
3 : $(m_0, m_1) \leftarrow \mathcal{A}.\mathsf{P}(\mathsf{pk})$
4 : $c \leftarrow \mathsf{Saber.Enc}(\mathsf{pk}, m_u)$
5 : $u' \leftarrow \mathcal{A}.\mathsf{D}(\mathsf{pk}, c)$
6 : **return** $(u' = u)$
```

Figure 3.2: The IND-CPA Game for Saber.PKE

Moreover, the corresponding advantage becomes the following.

$$\mathsf{Adv}^{\text{IND-CPA}}_{\text{Saber.PKE}}(\mathcal{A}) = \left|\Pr\left[\mathrm{Game}^{\text{IND-CPA}}_{\mathcal{A},\text{Saber.PKE}} = 1\right] - \frac{1}{2}\right|$$

Technically, we could construct a more detailed description of $\mathrm{Game}^{\text{IND-CPA}}_{\mathcal{A},\text{Saber.PKE}}$ by replacing the calls to Saber.PKE's algorithms with their actual specification; in fact, this is necessary to carry out the scheme's security proof properly. However, for reasons of conciseness, we do not yet perform this replacement here.

Concerning the relevant hardness assumptions, we formalize the MLWR problem as the MLWR game, which encapsulates the ability of an adversary to distinguish between a certain number of MLWR samples and the same number of uniformly random samples (from the same domain). Indeed, the assumption that no relevant adversary can successfully distinguish between these cases with non-negligible probability is equivalent to the assumption that the MLWR problem is hard. The description of the MLWR game is given in Figure 3.3, for adversary $\mathcal{A}$ and parameters $m, l, \mu, q,$ and p.

In contrast to the IND-CPA game, which samples u uniformly at random, the MLWR game takes this bit as input. As a result, the advantage of an adversary against the MLWR game can be formulated in a way that simplifies several of the derivations carried out in the security proof. Concretely, the advantage of an adversary $\mathcal{A}$ against $\mathrm{Game}^{\text{MLWR}}_{\mathcal{A},m,l,\mu,q,p}$ is defined as follows.

$$\mathsf{Adv}^{\text{MLWR}}_{m,l,\mu,q,p}(\mathcal{A}) = \left|\Pr\left[\mathrm{Game}^{\text{MLWR}}_{\mathcal{A},m,l,\mu,q,p}(1) = 1\right] - \Pr\left[\mathrm{Game}^{\text{MLWR}}_{\mathcal{A},m,l,\mu,q,p}(0) = 1\right]\right|$$

[5]Naturally, since the adversary must belong to the class of relevant adversaries, it is subject to the general restrictions imposed on this class.

```
Game$^{\text{MLWR}}_{\mathcal{A},m,l,\mu,q,p}(u)$
1 :  $\mathbf{A} \leftarrow\$ \ \mathcal{U}(R_q^{m\times l})$
2 :  $\mathbf{s} \leftarrow\$ \ \beta_\mu(R_q^{l\times 1})$
3 :  $\mathbf{b}_0 \leftarrow \lfloor \mathbf{A} \cdot \mathbf{s} \rceil_{q\to p}$
4 :  $\mathbf{b}_1 \leftarrow\$ \ \mathcal{U}(R_p^{m\times 1})$
5 :  return $\mathcal{A}(\mathbf{A}, \mathbf{b}_u)$
```

Figure 3.3: The MLWR Game

As desired, $\mathsf{Adv}^{\text{MLWR}}_{m,l,\mu,q,p}(\mathcal{A})$ precisely captures the extent to which $\mathcal{A}$ is capable of distinguishing between the case in which the game provides uniformly random samples, i.e., $u = 1$, and the case in which the game provides MLWR samples, i.e., $u = 0$. Namely, any success $\mathcal{A}$ has in distinguishing these cases is reflected in the difference between the probability of $\mathcal{A}$ returning a specific value in the case that $u = 1$ and the probability of $\mathcal{A}$ returning this same value in the case that $u = 0$. In $\mathsf{Adv}^{\text{MLWR}}_{m,l,\mu,q,p}(\mathcal{A})$, this specific value is arbitrarily chosen to be 1; indeed, altering this value to 0 for both probabilities yields an equivalent definition.

In lieu of the MLWR game specified above, the security proof of Saber.PKE utilizes a variant of this game; we refer to this variant as the "GMLWR game". Essentially, the GMLWR game deviates from the MLWR game in two aspects. First, instead of sampling $\mathbf{A}$ uniformly at random, the GMLWR game generates this matrix by evaluating gen on a uniformly random seed; here, gen is the same algorithm as the one employed in the specifications of $\mathsf{Saber.KeyGen}$ and $\mathsf{Saber.Enc}$. Second, rather than $\mathbf{A}$, the GMLWR game gives the adversary the seed utilized to generate $\mathbf{A}$. Since gen is assumed to be publicly known and efficiently evaluable, the adversary is capable of evaluating gen on this seed to generate $\mathbf{A}$; this implies that passing the seed provides the adversary with at least as much information as directly passing $\mathbf{A}$. With respect to adversary $\mathcal{A}$ and parameters $l, \mu, q,$ and p, the description of the GMLWR game is provided in Figure 3.4[6]. For clarity purposes, the naming of the parameters between the MLWR and GMLWR games is kept consistent; in particular, in both cases, the q and l parameters respectively denote the modulus in R_q and the dimension of the vectors/matrices. However, as opposed to the MLWR game, the GMLWR game is not defined with respect to the parameter m. The reason for this is that, as aforementioned, gen generates elements of $R_q^{l\times l}$; because these elements are square matrices, only a single parameter is needed to specify their dimension. Consequently, this indicates that the GMLWR game is actually a variant of the MLWR game with $m = l$, i.e., the MLWR game in which the number of samples equals the number of polynomial entries per sample.

```
Game$^{\text{GMLWR}}_{\mathcal{A},l,\mu,q,p}(u)$
1 :  $\text{seed}_{\mathbf{A}} \leftarrow\$ \ \mathcal{U}(\{0,1\}^{256})$
2 :  $\mathbf{A} \leftarrow \mathsf{gen}(\text{seed}_{\mathbf{A}})$
3 :  $\mathbf{s} \leftarrow\$ \ \beta_\mu(R_q^{l\times 1})$
4 :  $\mathbf{b}_0 \leftarrow \lfloor \mathbf{A} \cdot \mathbf{s} \rceil_{q\to p}$
5 :  $\mathbf{b}_1 \leftarrow\$ \ \mathcal{U}(R_p^{l\times 1})$
6 :  return $\mathcal{A}(\text{seed}_{\mathbf{A}}, \mathbf{b}_u)$
```

Figure 3.4: The GMLWR Game

[6]Technically, the GMLWR game would be valid for any seed length that gen could (be instantiated to) accept; however, the seed has a fixed length of 256 bits since this is the seed length used in Saber.PKE's algorithms.

Correspondingly, the advantage of an adversary $\mathcal{A}$ against $\text{Game}^{\text{GMLWR}}_{\mathcal{A},l,\mu,q,p}$ is defined as follows.

$$\mathsf{Adv}^{\text{GMLWR}}_{l,\mu,q,p}(\mathcal{A}) = \left|\Pr\left[\text{Game}^{\text{GMLWR}}_{\mathcal{A},l,\mu,q,p}(1) = 1\right] - \Pr\left[\text{Game}^{\text{GMLWR}}_{\mathcal{A},l,\mu,q,p}(0) = 1\right]\right|$$

In addition to the GMLWR game, the security proof of Saber.PKE utilizes a game which we refer to as the "XMLWR game"; basically, this game is a mixture of the MLWR and GMLWR games. More precisely, the XMLWR game constructs $\mathbf{A}$ and $\mathbf{b}_u$ similarly to the GMLWR game; however, the XMLWR game additionally creates a pair $(\mathbf{a}, d_u)$ in the same manner as the MLWR game creates its samples. Ultimately, the adversary is given the seed utilized to generate $\mathbf{A}$, $\mathbf{b}_u$, $\mathbf{a}$, and d_u. For adversary $\mathcal{A}$ and parameters $l, \mu, q,$ and p, the XMLWR game is defined in Figure 3.5[7].

```
$\text{Game}^{\text{XMLWR}}_{\mathcal{A},l,\mu,q,p}(u)$
1 : $\text{seed}_{\mathbf{A}} \leftarrow\$ \mathcal{U}(\{0,1\}^{256})$
2 : $\mathbf{A} \leftarrow \mathsf{gen}(\text{seed}_{\mathbf{A}})$
3 : $\mathbf{s} \leftarrow\$ \beta_\mu(R_q^{l\times 1})$
4 : $\mathbf{b}_0 \leftarrow \lfloor \mathbf{A}^T \cdot \mathbf{s} \rceil_{q\to p}$
5 : $\mathbf{b}_1 \leftarrow\$ \mathcal{U}(R_p^{l\times 1})$
6 : $\mathbf{a} \leftarrow\$ \mathcal{U}(R_q^{1\times l})$
7 : $d_0 \leftarrow \lfloor \mathbf{a} \cdot \mathbf{s} \rceil_{q\to p}$
8 : $d_1 \leftarrow\$ \mathcal{U}(R_p)$
9 : return $\mathcal{A}(\text{seed}_{\mathbf{A}}, \mathbf{b}_u, \mathbf{a}, d_u)$
```

Figure 3.5: The XMLWR Game

Accordingly, the advantage of an adversary $\mathcal{A}$ against $\text{Game}^{\text{XMLWR}}_{\mathcal{A},l,\mu,q,p}$ game is defined below.

$$\mathsf{Adv}^{\text{XMLWR}}_{l,\mu,q,p}(\mathcal{A}) = \left|\Pr\left[\text{Game}^{\text{XMLWR}}_{\mathcal{A},l,\mu,q,p}(1) = 1\right] - \Pr\left[\text{Game}^{\text{XMLWR}}_{\mathcal{A},l,\mu,q,p}(0) = 1\right]\right|$$

Lastly, due to the similarity between the MLWR game and the GMLWR/XMLWR game, the hardness of the GMLWR/XMLWR game is related to the hardness of the MLWR game. Intuitively, if the output distribution of gen (closely) resembles $\mathcal{U}(R_q^{l\times l})$ and gen has no structure that can be exploited to predict its output values, the GMLWR/XMLWR game is (nearly) as hard as the MLWR game. The subsequent discussion further elaborates on this intuition and, afterward, formalizes this intuition through ROM proofs.

Security Theorem

The following security theorem relates the IND-CPA security of Saber.PKE to the problems associated with the GMLWR and XMLWR games. Specifically, the theorem provides an upper bound on the advantage that an adversary can achieve against $\text{Game}^{\text{IND-CPA}}_{\mathcal{A},\text{Saber.PKE}}$; this upper bound is expressed in terms of advantages against instances of the GMLWR and XMLWR games, each concretized with the parameters from Saber. That is, although $l, \mu, q,$ and p denote generic formal parameters in the definitions of the GMLWR game, the XMLWR game, and the advantages of adversaries against these games, the parameters used in the security theorem below directly correspond to the parameters of Saber; indeed, for reasons of consistency and clarity, these happen to have the same identifiers.

[7]For the XMLWR game, the same comments hold regarding the naming of the parameters and the seed length as for the GMLWR game. Likewise, the XMLWR game is not defined with respect to the parameter m for reasons identical to the ones provided for the GMLWR game.

Security Theorem. *Let $\frac{q}{p} \leq \frac{p}{2t}$. Then, for any adversary $\mathcal{A}$, there exist adversaries $\mathcal{B}_0$ and $\mathcal{B}_1$, each with approximately the same*[8] *running time as $\mathcal{A}$, such that*

$$\mathsf{Adv}^{\text{IND-CPA}}_{\text{Saber.PKE}}(\mathcal{A}) \leq \mathsf{Adv}^{\text{GMLWR}}_{l,\mu,q,p}(\mathcal{B}_0) + \mathsf{Adv}^{\text{XMLWR}}_{l,\mu,q,p}(\mathcal{B}_1)$$

Notably, this theorem does not impose any resource-related restrictions on $\mathcal{A}$, nor does it specify any requirements for the instantiation of gen; as such, the theorem essentially claims that its statement is veracious for *any* $\mathcal{A}$ with an arbitrary amount of resources and *any* instantiation of gen. Albeit we show this is indeed the case, the theorem is meaningful in practice only if the sum of $\mathsf{Adv}^{\text{GMLWR}}_{l,\mu,q,p}(\mathcal{B}_0)$ and $\mathsf{Adv}^{\text{XMLWR}}_{l,\mu,q,p}(\mathcal{B}_1)$ is guaranteed to be negligibly small for all relevant adversaries $\mathcal{A}$ against $\text{Game}^{\text{IND-CPA}}_{\mathcal{A},\text{Saber.PKE}}$. For instance, in case there exists a relevant adversary $\mathcal{A}$ against $\text{Game}^{\text{IND-CPA}}_{\mathcal{A},\text{Saber.PKE}}$ such that $\mathsf{Adv}^{\text{GMLWR}}_{l,\mu,q,p}(\mathcal{B}_0) + \mathsf{Adv}^{\text{XMLWR}}_{l,\mu,q,p}(\mathcal{B}_1) = \alpha$ for some non-negligibly small α, the theorem's inequality reduces to $\mathsf{Adv}^{\text{IND-CPA}}_{\text{Saber.PKE}}(\mathcal{A}) \leq \alpha$; this does not ensure that $\mathsf{Adv}^{\text{IND-CPA}}_{\text{Saber.PKE}}(\mathcal{A})$ is negligibly small. Consequently, in this case, the theorem does not sufficiently limit the potential advantage of this relevant adversary against $\text{Game}^{\text{IND-CPA}}_{\mathcal{A},\text{Saber.PKE}}$ and, hence, we cannot conclude that Saber.PKE is IND-CPA secure. Conversely, suppose that for any relevant adversary $\mathcal{A}$ against $\text{Game}^{\text{IND-CPA}}_{\mathcal{A},\text{Saber.PKE}}$, we have $\mathsf{Adv}^{\text{GMLWR}}_{l,\mu,q,p}(\mathcal{B}_0) + \mathsf{Adv}^{\text{XMLWR}}_{l,\mu,q,p}(\mathcal{B}_1) \leq \epsilon$ such that ϵ is negligibly small; then, from the theorem's inequality, it follows that $\mathsf{Adv}^{\text{IND-CPA}}_{\text{Saber.PKE}}(\mathcal{A}) \leq \epsilon$ for any relevant adversary $\mathcal{A}$. That is, in this situation, the theorem's inequality guarantees that the advantage of any relevant adversary $\mathcal{A}$ against $\text{Game}^{\text{IND-CPA}}_{\mathcal{A},\text{Saber.PKE}}$ is negligibly small; indeed, this implies the IND-CPA security of Saber.PKE.

For $\mathsf{Adv}^{\text{GMLWR}}_{l,\mu,q,p}(\mathcal{B}_0) + \mathsf{Adv}^{\text{XMLWR}}_{l,\mu,q,p}(\mathcal{B}_1)$ to be negligibly small, $\mathsf{Adv}^{\text{GMLWR}}_{l,\mu,q,p}(\mathcal{B}_0)$ and $\mathsf{Adv}^{\text{XMLWR}}_{l,\mu,q,p}(\mathcal{B}_1)$ must individually be negligibly small as well; in turn, this requires the corresponding GMLWR and XMLWR games to be hard to win[9]. The hardness of these games is an assumption that we make and corroborate for appropriate parameters and instantiations of gen. Namely, intuitively, given that the MLWR game (with $m = l$) is conjectured to be hard for suitable parameter choices, it seems reasonable to assume that the GMLWR and XMLWR games are hard for comparable parameter choices *if* gen is adequately instantiated. Specifically, if the output distribution of gen (closely) resembles $\mathcal{U}(R_q^{l \times l})$ and gen has (practically) no structure that can be exploited to predict its output values, i.e., gen is (effectively) a random oracle, $\text{Game}^{\text{GMLWR}}_{\mathcal{B}_0,l,\mu,q,p}$ is essentially equal to $\text{Game}^{\text{MLWR}}_{\mathcal{B}_0,l,l,\mu,q,p}$. Moreover, the XMLWR game is similar to the GMLWR game with one additional sample generated as in the MLWR game; consequently, for a satisfactory instantiation of gen, $\text{Game}^{\text{XMLWR}}_{\mathcal{B}_1,l,\mu,q,p}$ is virtually the same as $\text{Game}^{\text{MLWR}}_{\mathcal{B}_1,l+1,l,\mu,q,p}$.

Hardness of GMLWR and XMLWR

Albeit the preceding intuitive substantiation for the hardness of the GMLWR and XMLWR games might seem correct and convincing, we formalize the presented arguments by means of ROM proofs; in essence, these proofs show that if gen is a random oracle and the MLWR game is hard, then the GMLWR and XMLWR games are hard. Specifically, assuming gen is a random oracle, we show that any instance of the MLWR game efficiently reduces to corresponding instances of the GMLWR and XMLWR games. Alternatively stated, given an adversary $\mathcal{A}$ against any instance of the GMLWR/XMLWR game in which gen is a random oracle, we construct an adversary against a corresponding instance of the MLWR game such that this constructed adversary has the same advantage in (the instance of) the MLWR game as $\mathcal{A}$ has in (the instance of) the GMLWR/XMLWR

[8]Generally, "approximately the same" and (correspondingly) "efficient" are formally defined depending on the employed approach to provable security; for example, "within a polynomial factor" and, respectively, "polynomial running time" are common such definitions. Nevertheless, in this thesis, the discussions and reasoning that utilize these terms are trivially valid for any customary definitions. As such, we refrain from fixing a specific definition and merely refer to the abstract concepts.

[9]Recall that, as suggested in Section 2.2, "hard" and "hardness" are invariably defined with respect to a class of efficient or resource-restrained adversaries, i.e., adversaries that do *not* have an arbitrary amount of resources.

game; moreover, this constructed adversary has approximately the same running time as $\mathcal{A}$. Then, the hardness of the GMLWR and XMLWR games follows from the conjectured hardness of the MLWR game. Namely, according to this hardness assumption, no relevant adversary against the MLWR game can achieve a non-negligible advantage[10]; nevertheless, if gen is a random oracle, we show that we can create such an adversary from an adversary that achieves a non-negligible advantage against the GMLWR/XMLWR game. Therefore, such adversaries against the GMLWR and XMLWR games must also not exist, i.e., the GMLWR and XMLWR games must be hard (assuming gen is a random oracle).

```
$\text{GameROM}^{\text{GMLWR}}_{\mathcal{A},l,\mu,q,p}(u)$
1 : $\text{seed}_{\mathbf{A}} \leftarrow\$ \mathcal{U}(\{0,1\}^{256})$
2 : $\mathbf{A} \leftarrow \mathsf{Gen}(\text{seed}_{\mathbf{A}})$
3 : $\mathbf{s} \leftarrow\$ \beta_\mu(R_q^{l\times 1})$
4 : $\mathbf{b}_0 \leftarrow \lfloor \mathbf{A}\cdot\mathbf{s} \rceil_{q\to p}$
5 : $\mathbf{b}_1 \leftarrow\$ \mathcal{U}(R_p^{l\times 1})$
6 : $\textbf{return } \mathcal{A}^{\mathsf{Gen}}(\text{seed}_{\mathbf{A}}, \mathbf{b}_u)$
```

```
$\text{GameROM}^{\text{XMLWR}}_{\mathcal{A},l,\mu,q,p}(u)$
1 : $\text{seed}_{\mathbf{A}} \leftarrow\$ \mathcal{U}(\{0,1\}^{256})$
2 : $\mathbf{A} \leftarrow \mathsf{Gen}(\text{seed}_{\mathbf{A}})$
3 : $\mathbf{s} \leftarrow\$ \beta_\mu(R_q^{l\times 1})$
4 : $\mathbf{b}_0 \leftarrow \lfloor \mathbf{A}^T\cdot\mathbf{s} \rceil_{q\to p}$
5 : $\mathbf{b}_1 \leftarrow\$ \mathcal{U}(R_p^{l\times 1})$
6 : $\mathbf{a} \leftarrow\$ \mathcal{U}(R_q^{1\times l})$
7 : $d_0 \leftarrow \lfloor \mathbf{a}\cdot\mathbf{s} \rceil_{q\to p}$
8 : $d_1 \leftarrow\$ \mathcal{U}(R_p)$
9 : $\textbf{return } \mathcal{A}^{\mathsf{Gen}}(\text{seed}_{\mathbf{A}}, \mathbf{b}_u, \mathbf{a}, d_u)$
```

Figure 3.6: The GMLWR (Left) and XMLWR (Right) Games in the Random Oracle Model

Foremost, to differentiate between the standard model and random oracle model, we introduce separate identifiers and descriptions for the GMLWR and XMLWR games in the ROM; moreover, in these descriptions, to distinguish between the standard gen and its idealized counterpart, we denote the latter by Gen. For adversary $\mathcal{A}$ and parameters $l, \mu, q,$ and p, Figure 3.6 provides the descriptions of these GMLWR and XMLWR games in the ROM.

As above-mentioned, for the reductions from MLWR to GMLWR and XMLWR, we aspire to construct an adversary against the MLWR game from a given adversary against the GMLWR game and, respectively, the XMLWR game; additionally, in both cases, this constructed reduction adversary must have a running time that is approximately the same as the given adversary's running time. An essential component in these reductions is the reduction adversary's ability to employ the given adversary's algorithms as well as monitor and control all of the corresponding inputs and outputs. Particularly, a reduction adversary can monitor and manipulate each random oracle query issued by the given adversary. Nevertheless, to guarantee that the given adversary behaves identically between the reduction and a regular run of its own game, the reduction adversary must ensure that the given adversary cannot distinguish between these cases. In this context, this comes down to ensuring that the input provided to the given adversary in the reduction is indistinguishable from the input in a run of its own game; moreover, the distribution of the manipulated random oracle query results must be indistinguishable from the regular distribution of these query results, i.e., the uniform distribution.

Regarding the reduction from MLWR to GMLWR, consider an adversary $\mathcal{A}$ against $\text{GameROM}^{\text{GMLWR}}_{\mathcal{A},l,\mu,q,p}$. Given this adversary, we can straightforwardly construct an adversary $\mathcal{R}^{\mathcal{A}}$ (i.e., $\mathcal{R}$ can use $\mathcal{A}$ as a black box sub-procedure) against $\text{Game}^{\text{MLWR}}_{\mathcal{R}^{\mathcal{A}},l,l,\mu,q,p}$. Namely, comparing these two games, we see that they are nearly identical; indeed, the sole differences between these games concern the manner in which $\mathbf{A}$ is obtained and the information passed to the adversary. However, since Gen is a random oracle, the $\mathbf{A}$ in $\text{GameROM}^{\text{GMLWR}}_{\mathcal{A},l,\mu,q,p}$ is uniformly distributed over $R_q^{l\times l}$, similarly to

[10]Technically, as alluded to before, the MLWR game is only considered hard for suitable instances of the game. However, we construct abstract reductions covering all possible instances; indeed, this encompasses practically appropriate instances for which the game is actually considered hard.

its counterpart in $\text{Game}^{\text{MLWR}}_{\mathcal{A},l,l,\mu,q,p}$. Consequently, the way in which $\mathbf{A}$ is acquired is equivalent between these two games. Following, the only actual difference between the games is that, while $\text{Game}^{\text{MLWR}}_{\mathcal{A},l,l,\mu,q,p}$ directly gives $\mathbf{A}$ to its adversary, $\text{GameROM}^{\text{GMLWR}}_{\mathcal{A},l,\mu,q,p}$ provides the seed with which Gen is queried to obtain $\mathbf{A}$. Combining these observations, we construct the above-mentioned adversary $\mathcal{R}^{\mathcal{A}}$ against $\text{Game}^{\text{MLWR}}_{\mathcal{R}^{\mathcal{A}},l,l,\mu,q,p}$ as follows.

1. Upon being called by $\text{Game}^{\text{MLWR}}_{\mathcal{R}^{\mathcal{A}},l,l,\mu,q,p}$, $\mathcal{R}^{\mathcal{A}}$ stores the given parameters $\mathbf{A}$ and $\mathbf{b}_u$.

2. Afterward, $\mathcal{R}^{\mathcal{A}}$ samples a seed uniformly at random from $\mathcal{U}(\{0,1\}^{256})$; that is, $\mathcal{R}^{\mathcal{A}}$ performs $\text{seed}_{\mathbf{A}} \leftarrow\!\$\ \mathcal{U}(\{0,1\}^{256})$.

3. Then, $\mathcal{R}^{\mathcal{A}}$ calls $\mathcal{A}(\text{seed}_{\mathbf{A}}, \mathbf{b}_u)$ and proceeds to monitor all random oracle queries. If $\mathcal{A}$ queries the random oracle on $\text{seed}_{\mathbf{A}}$, $\mathcal{R}^{\mathcal{A}}$ blocks the query and returns $\mathbf{A}$; otherwise, $\mathcal{R}^{\mathcal{A}}$ allows the random oracle to answer the query.

4. Lastly, $\mathcal{R}^{\mathcal{A}}$ directly returns the value retrieved from $\mathcal{A}(\text{seed}_{\mathbf{A}}, \mathbf{b}_u)$.

Naturally, fixing the response for a single random oracle query with a uniformly distributed value, as is done by $\mathcal{R}^{\mathcal{A}}$, does not alter the distribution of the random oracle query results. As a result, $\mathcal{R}^{\mathcal{A}}$ perfectly simulates a run of $\mathcal{A}$'s game, i.e., $\text{GameROM}^{\text{GMLWR}}_{\mathcal{A},l,\mu,q,p}$, using the values provided by its own game, i.e., $\text{Game}^{\text{MLWR}}_{\mathcal{R}^{\mathcal{A}},l,l,\mu,q,p}$. Hence, $\mathcal{A}$ is guaranteed to behave as in a run of its own game; additionally, $\mathcal{A}$ attempts to solve the same problem as $\mathcal{R}^{\mathcal{A}}$. In turn, this means that the reduction adversary successfully employs $\mathcal{A}$ to obtain an advantage against $\text{Game}^{\text{MLWR}}_{\mathcal{R}^{\mathcal{A}},l,l,\mu,q,p}$ that is equal to the advantage of $\mathcal{A}$ against $\text{GameROM}^{\text{GMLWR}}_{\mathcal{A},l,\mu,q,p}$. Furthermore, excluding the call to $\mathcal{A}(\text{seed}_{\mathbf{A}}, \mathbf{b}_u)$, $\mathcal{R}^{\mathcal{A}}$ solely performs sequential operations that can trivially be executed efficiently; as such, it is evident that the running time of $\mathcal{R}^{\mathcal{A}}$ is approximately the same as the running time of $\mathcal{A}$. Concluding, the preceding constitutes a correct and efficient reduction from the MLWR game to the GMLWR game in the ROM; that is, if gen is a random oracle and the MLWR game is hard, then the GMLWR game is hard as well.

Finally, concerning the reduction from MLWR to XMLWR, we construct an adversary $\mathcal{R}^{\mathcal{A}}$ against $\text{Game}^{\text{MLWR}}_{\mathcal{R}^{\mathcal{A}},l+1,l,\mu,q,p}$ from an adversary $\mathcal{A}$ against $\text{GameROM}^{\text{XMLWR}}_{\mathcal{A},l,\mu,q,p}$. In this reduction, we utilize the following two mathematical observations. First, a matrix-vector multiplication essentially computes a series of inner products, orderly storing the results in a vector. Specifically, the vector resulting from a matrix-vector multiplication comprises the inner products of the matrix's row vectors with the multiplication's operand vector. Second, if a matrix is uniformly distributed, each of its rows is also uniformly distributed; moreover, removing a row from a uniformly distributed matrix gives another uniformly distributed matrix. Applying these observations to the current context, we deduce that extracting a row from the matrix $\mathbf{A}$ in $\text{Game}^{\text{MLWR}}_{\mathcal{R}^{\mathcal{A}},l+1,l,\mu,q,p}$, which is uniformly distributed over $R_q^{l+1\times l}$, produces a matrix and a vector that are uniformly distributed over $R_q^{l\times l}$ and $R_q^{1\times l}$, respectively. Utilizing this result, we construct adversary $\mathcal{R}^{\mathcal{A}}$ against $\text{Game}^{\text{MLWR}}_{\mathcal{R}^{\mathcal{A}},l+1,l,\mu,q,p}$ as follows.

1. Upon being called by $\text{Game}^{\text{MLWR}}_{\mathcal{R}^{\mathcal{A}},l+1,l,\mu,q,p}$, $\mathcal{R}^{\mathcal{A}}$ respectively extracts the last row and entry from the given parameters $\mathbf{A}$ and $\mathbf{b}_u$; subsequently, it stores the four resulting artifacts. For convenience, we accordingly refer to the matrix and vector produced by the row extraction as $\mathbf{A}'$ and $\mathbf{b}'_u$; similarly, we denote the vector and polynomial resulting from the entry extraction by, respectively, $\mathbf{a}$ and d_u.

2. Afterward, $\mathcal{R}^{\mathcal{A}}$ samples a seed uniformly at random from $\mathcal{U}(\{0,1\}^{256})$; that is, $\mathcal{R}^{\mathcal{A}}$ performs $\text{seed}_{\mathbf{A}'} \leftarrow\!\$\ \mathcal{U}(\{0,1\}^{256})$.

3. Then, $\mathcal{R}^{\mathcal{A}}$ calls $\mathcal{A}(\text{seed}_{\mathbf{A}'}, \mathbf{b}'_u, \mathbf{a}, d_u)$ and continues to monitor all random oracle queries. In case $\mathcal{A}$ queries the random oracle on $\text{seed}_{\mathbf{A}'}$, $\mathcal{R}^{\mathcal{A}}$ blocks the query and returns $\mathbf{A}'^T$; otherwise, $\mathcal{R}^{\mathcal{A}}$ allows the random oracle to answer the query.

4. Lastly, $\mathcal{R}^{\mathcal{A}}$ directly returns the value retrieved from $\mathcal{A}(\text{seed}_{\mathbf{A}'}, \mathbf{b}'_u, \mathbf{a}, d_u)$.

Here, when $\mathcal{A}$ queries the random oracle on $\text{seed}_{\mathbf{A}'}$, $\mathcal{R}^{\mathcal{A}}$ returns $\mathbf{A}'^T$ instead of $\mathbf{A}'$ in order to compensate for the deviating computations of $\mathbf{b}_u$ between the MLWR and XMLWR games. Namely, since $\mathcal{A}$ is an adversary against the XMLWR game, it expects the reduction's $\mathbf{b}'_u$ to be computed with the transposed of the matrix obtained by querying Gen on $\text{seed}_{\mathbf{A}'}$. As such, since this $\mathbf{b}'_u$ is actually computed with $\mathbf{A}'$, $\mathcal{R}^{\mathcal{A}}$ must return $\mathbf{A}'^T$ to match $\mathcal{A}$'s expectations. Combining this with the previously discussed observations, we see that $\mathcal{R}^{\mathcal{A}}$ perfectly simulates a run of $\mathcal{A}$'s game using the values from its own game. Moreover, akin to before, fixing the response for a single random oracle query with a uniformly distributed value, as is done in this reduction, does not alter the distribution of the random oracle query results. Hence, similarly to the preceding reduction, it follows that $\mathcal{R}^{\mathcal{A}}$ successfully utilizes $\mathcal{A}$ to achieve an advantage against $\text{Game}^{\text{MLWR}}_{\mathcal{R}^{\mathcal{A}},l+1,l,\mu,q,p}$ that is identical to the advantage of $\mathcal{A}$ against $\text{GameROM}^{\text{GMLWR}}_{\mathcal{A},l,\mu,q,p}$. Furthermore, apart from the call to $\mathcal{A}(\text{seed}_{\mathbf{A}}, \mathbf{b}_u)$, $\mathcal{R}^{\mathcal{A}}$ exclusively performs sequential operations that can straightforwardly be executed efficiently; consequently, the running time of $\mathcal{R}^{\mathcal{A}}$ is approximately the same as that of $\mathcal{A}$. In conclusion, the preceding constitutes a correct and efficient reduction from the MLWR game to the XMLWR game in the ROM; as a result, if gen is a random oracle and the MLWR game is hard, then the XMLWR game is hard as well.

IND-CPA **Security of** Saber.PKE

Leveraging the defined IND-CPA, GMLWR, and XMLWR games, we presently prove the above-discussed security theorem through a code-based, game-playing security proof.

In total, the game-playing proof of the security theorem comprises five games. The initial game, $\text{Game}^0_{\mathcal{A}}$, is identical to the aforementioned $\text{Game}^{\text{IND-CPA}}_{\mathcal{A},\text{Saber.PKE}}$ after replacing the calls to $\mathsf{Saber.KeyGen}$ and $\mathsf{Saber.Enc}$ by their actual specification. As such, $\text{Game}^0_{\mathcal{A}}$ exactly captures the IND-CPA security of Saber.PKE. Figure 3.7 provides the definition of $\text{Game}^0_{\mathcal{A}}$ extended with comments (i.e., lines starting with '//') highlighting the different phases of the IND-CPA game.

Starting from $\text{Game}^0_{\mathcal{A}}$, the proof advances in incremental steps until it reaches the final game, $\text{Game}^4_{\mathcal{A}}$. In each of these steps, we make a slight adjustment to the currently considered game, producing the next game in the sequence. The differences between the original and resulting game of a step are employed as the basis of a reduction; specifically, for a step from $\text{Game}^i_{\mathcal{A}}$ to $\text{Game}^{i+1}_{\mathcal{A}}$, we show one of the following.

- Any adversary against $\text{Game}^i_{\mathcal{A}}$ can be utilized to construct an adversary against $\text{Game}^{i+1}_{\mathcal{A}}$.

- Any adversary distinguishing between $\text{Game}^i_{\mathcal{A}}$ and $\text{Game}^{i+1}_{\mathcal{A}}$ can be utilized to construct an adversary against an instance of the GMLWR or XMLWR game. Here, "distinguishing" refers to the ability of an adversary to achieve a different winning probability between the considered games; more formally, considering adversary $\mathcal{A}$, $\text{Game}^i_{\mathcal{A}}$, and $\text{Game}^{i+1}_{\mathcal{A}}$, the adversary is said to distinguish between the two games if $\left|\Pr\left[\text{Game}^i_{\mathcal{A}} = 1\right] - \Pr\left[\text{Game}^{i+1}_{\mathcal{A}} = 1\right]\right| > 0$.

Moreover, since all $\text{Game}^i_{\mathcal{A}}$ in the proof are variants of $\text{Game}^{\text{IND-CPA}}_{\mathcal{A},\text{Saber.PKE}}$, their advantage is similarly defined as follows.

$$\mathsf{Adv}^i(\mathcal{A}) = \left|\Pr\left[\text{Game}^i_{\mathcal{A}} = 1\right] - \frac{1}{2}\right|$$

Figure 3.8 depicts the definition of the complete game sequence (i.e., $\text{Game}^0_{\mathcal{A}}$ to $\text{Game}^4_{\mathcal{A}}$); in

```
$\mathrm{Game}^0_{\mathcal{A}} := \mathrm{Game}^{\text{IND-CPA}}_{\mathcal{A},\text{Saber.PKE}}$
1 :  $u \leftarrow\$ \mathcal{U}(\{0,1\})$
2 :  // — Key Generation ($\mathsf{Saber.KeyGen}$) —
3 :  $\mathrm{seed}_{\mathbf{A}} \leftarrow\$ \mathcal{U}(\{0,1\}^{256})$
4 :  $\mathbf{A} \leftarrow \mathsf{gen}(\mathrm{seed}_{\mathbf{A}})$
5 :  $\mathbf{s} \leftarrow\$ \beta_\mu(R_q^{l\times 1})$
6 :  $\mathbf{b} \leftarrow \lfloor \mathbf{A}\cdot\mathbf{s}+\mathbf{h}\rceil_{q\to p}$
7 :  // Key Pair: $(\mathsf{pk} = (\mathrm{seed}_{\mathbf{A}}, \mathbf{b}), \mathsf{sk} = \mathbf{s})$
8 :  // — Adversary Message Selection —
9 :  $(m_0, m_1) \leftarrow \mathcal{A}.\mathsf{P}((\mathrm{seed}_{\mathbf{A}}, \mathbf{b}))$
10 : // — Encryption ($\mathsf{Saber.Enc}$) —
11 : $\mathbf{s}' \leftarrow\$ \beta_\mu(R_q^{l\times 1})$;
12 : $\mathbf{b}' \leftarrow \lfloor \mathbf{A}^T\cdot\mathbf{s}'+\mathbf{h}\rceil_{q\to p}$
13 : $v' \leftarrow \mathbf{b}^T\cdot(\mathbf{s}' \bmod p) + (h_1 \bmod p)$
14 : $\hat{c} \leftarrow \lfloor v' + \lfloor m_u\rceil_{2\to p}\rceil_{p\to 2\cdot t}$
15 : // Ciphertext: $c = (\hat{c}, \mathbf{b}')$
16 : // — Adversary Guess —
17 : $u' \leftarrow \mathcal{A}.\mathsf{D}((\mathrm{seed}_{\mathbf{A}}, \mathbf{b}), (\hat{c}, \mathbf{b}'))$
18 : **return** $(u' = u)$
```

Figure 3.7: Initial Game in the Game-Playing Proof of Saber.PKE

addition, for each game, the lines that differ from the preceding game are highlighted with a gray background.

Step 1: $\mathrm{Game}^0_{\mathcal{A}}$ **-** $\mathrm{Game}^1_{\mathcal{A}}$ In the first step, we alter the way in which $\mathbf{b}$ is obtained. Specifically, rather than computing $\mathbf{b}$ by $\lfloor \mathbf{A}\cdot\mathbf{s}+\mathbf{h}\rceil_{q\to p}$, as $\mathrm{Game}^0_{\mathcal{A}}$ does, $\mathrm{Game}^1_{\mathcal{A}}$ samples $\mathbf{b}$ uniformly at random from its domain. As a side-effect of this change, $\mathrm{Game}^1_{\mathcal{A}}$ does not utilize $\mathbf{s}$ anymore; for this reason, $\mathbf{s}$ is completely removed from $\mathrm{Game}^1_{\mathcal{A}}$.

Considering the difference between $\mathrm{Game}^0_{\mathcal{A}}$ and $\mathrm{Game}^1_{\mathcal{A}}$, we see that the pair $(\mathbf{A}, \mathbf{b})$ comprises l GMLWR samples in $\mathrm{Game}^0_{\mathcal{A}}$; contrarily, in $\mathrm{Game}^1_{\mathcal{A}}$, this pair is uniform over its domain. Consequently, an adversary $\mathcal{A}$ that is able to distinguish between these two games can be used to construct an adversary $\mathcal{B}^{\mathcal{A}}_0$ against the GMLWR game. Figure 3.9 provides such a reduction. To make the connection with $\mathrm{Game}^0_{\mathcal{A}}$ more explicit, the fourth line in this reduction differs from the corresponding line in the original description of the GMLWR game (see Figure 3.4); particularly, instead of the $\mathbf{b}_0 \leftarrow \lfloor \mathbf{A}\cdot\mathbf{s}\rceil_{q\to p}$ statement from the original description, the reduction uses $\mathbf{b}_0 \leftarrow \lfloor \mathbf{A}\cdot\mathbf{s}+\mathbf{h}\rceil_{q\to p}$. Nevertheless, as shown in Section 3.1, these two computations are equivalent, making them interchangeable.

Based on the reduction given in Figure 3.9, we can determine that for any given adversary $\mathcal{A}$ against $\mathrm{Game}^1_{\mathcal{A}}$ and $\mathrm{Game}^2_{\mathcal{A}}$, there exists an adversary $\mathcal{B}^{\mathcal{A}}_0$ against the corresponding instance of the GMLWR game such that $\left|\Pr\left[\mathrm{Game}^0_{\mathcal{A}} = 1\right] - \Pr\left[\mathrm{Game}^1_{\mathcal{A}} = 1\right]\right| = \mathsf{Adv}^{\mathrm{GMLWR}}_{l,\mu,q,p}(\mathcal{B}^{\mathcal{A}}_0)$. Specifically, this result can be deduced as follows.

$$\begin{aligned}
\forall_{\mathcal{A}}\exists_{\mathcal{B}^{\mathcal{A}}_0} : \mathsf{Adv}^{\mathrm{GMLWR}}_{l,\mu,q,p}(\mathcal{B}^{\mathcal{A}}_0) &= \left|\Pr\left[\mathrm{Game}^{\mathrm{GMLWR}}_{\mathcal{B}^{\mathcal{A}}_0,l,\mu,q,p}(1) = 1\right] - \Pr\left[\mathrm{Game}^{\mathrm{GMLWR}}_{\mathcal{B}^{\mathcal{A}}_0,l,\mu,q,p}(0) = 1\right]\right| \\
&= \left|\Pr[w' = w \mid u = 1] - \Pr[w' = w \mid u = 0]\right| \\
&= \left|\Pr\left[\mathrm{Game}^1_{\mathcal{A}} = 1\right] - \Pr\left[\mathrm{Game}^0_{\mathcal{A}} = 1\right]\right| \\
&= \left|\Pr\left[\mathrm{Game}^0_{\mathcal{A}} = 1\right] - \Pr\left[\mathrm{Game}^1_{\mathcal{A}} = 1\right]\right|
\end{aligned}$$

$\text{Game}^0_{\mathcal{A}}$

```
1 :  $u \leftarrow\$ \mathcal{U}(\{0,1\})$
2 :  $\text{seed}_{\mathbf{A}} \leftarrow\$ \mathcal{U}(\{0,1\}^{256})$
3 :  $\mathbf{A} \leftarrow \mathsf{gen}(\text{seed}_{\mathbf{A}})$
4 :  $\mathbf{s} \leftarrow\$ \beta_\mu(R_q^{l\times 1})$
5 :  $\mathbf{b} \leftarrow \lfloor \mathbf{A}\cdot\mathbf{s}+\mathbf{h} \rceil_{q\to p}$
6 :  $(m_0, m_1) \leftarrow \mathcal{A}.\mathsf{P}((\text{seed}_{\mathbf{A}}, \mathbf{b}))$
7 :  $\mathbf{s}' \leftarrow\$ \beta_\mu(R_q^{l\times 1})$
8 :  $\mathbf{b}' \leftarrow \lfloor \mathbf{A}^T\cdot\mathbf{s}'+\mathbf{h} \rceil_{q\to p}$
9 :  $v' \leftarrow \mathbf{b}^T\cdot(\mathbf{s}' \bmod p) + (h_1 \bmod p)$
10 : $\hat{c} \leftarrow \lfloor v' + \lfloor m_u \rceil_{2\to p} \rceil_{p\to 2\cdot t}$
11 : $u' \leftarrow \mathcal{A}.\mathsf{D}((\text{seed}_{\mathbf{A}}, \mathbf{b}), (\hat{c}, \mathbf{b}'))$
12 : $\textbf{return } (u' = u)$
```

$\text{Game}^1_{\mathcal{A}}$

```
1 :  $u \leftarrow\$ \mathcal{U}(\{0,1\})$
2 :  $\text{seed}_{\mathbf{A}} \leftarrow\$ \mathcal{U}(\{0,1\}^{256})$
3 :  $\mathbf{A} \leftarrow \mathsf{gen}(\text{seed}_{\mathbf{A}})$
4 :  Skip
5 :  $\mathbf{b} \leftarrow\$ \mathcal{U}(R_p^{l\times 1})$
6 :  $(m_0, m_1) \leftarrow \mathcal{A}.\mathsf{P}((\text{seed}_{\mathbf{A}}, \mathbf{b}))$
7 :  $\mathbf{s}' \leftarrow\$ \beta_\mu(R_q^{l\times 1})$
8 :  $\mathbf{b}' \leftarrow \lfloor \mathbf{A}^T\cdot\mathbf{s}'+\mathbf{h} \rceil_{q\to p}$
9 :  $v' \leftarrow \mathbf{b}^T\cdot(\mathbf{s}' \bmod p) + (h_1 \bmod p)$
10 : $\hat{c} \leftarrow \lfloor v' + \lfloor m_u \rceil_{2\to p} \rceil_{p\to 2\cdot t}$
11 : $u' \leftarrow \mathcal{A}.\mathsf{D}((\text{seed}_{\mathbf{A}}, \mathbf{b}), (\hat{c}, \mathbf{b}'))$
12 : $\textbf{return } (u' = u)$
```

$\text{Game}^2_{\mathcal{A}}$

```
1 :  $u \leftarrow\$ \mathcal{U}(\{0,1\})$
2 :  $\text{seed}_{\mathbf{A}} \leftarrow\$ \mathcal{U}(\{0,1\}^{256})$
3 :  $\mathbf{A} \leftarrow \mathsf{gen}(\text{seed}_{\mathbf{A}})$
4 :  Skip
5 :  $\mathbf{b} \leftarrow\$ \mathcal{U}(R_p^{l\times 1})$
6 :  $(m_0, m_1) \leftarrow \mathcal{A}.\mathsf{P}((\text{seed}_{\mathbf{A}}, \mathbf{b}))$
7 :  $\mathbf{s}' \leftarrow\$ \beta_\mu(R_q^{l\times 1})$
8 :  $\mathbf{b}' \leftarrow \lfloor \mathbf{A}^T\cdot\mathbf{s}'+\mathbf{h} \rceil_{q\to p}$
9 :  $v' \leftarrow \mathbf{b}^T\cdot(\mathbf{s}' \bmod p) + (h_1 \bmod p)$
10 : $\hat{c} \leftarrow \lfloor v' + \lfloor m_u \rceil_{2\to p} \rceil_{p\to p^2/q}$
11 : $u' \leftarrow \mathcal{A}.\mathsf{D}((\text{seed}_{\mathbf{A}}, \mathbf{b}), (\hat{c}, \mathbf{b}'))$
12 : $\textbf{return } (u' = u)$
```

$\text{Game}^3_{\mathcal{A}}$

```
1 :  $u \leftarrow\$ \mathcal{U}(\{0,1\})$
2 :  $\text{seed}_{\mathbf{A}} \leftarrow\$ \mathcal{U}(\{0,1\}^{256})$
3 :  $\mathbf{A} \leftarrow \mathsf{gen}(\text{seed}_{\mathbf{A}})$
4 :  Skip
5 :  $\mathbf{b} \leftarrow\$ \mathcal{U}(R_q^{l\times 1})$
6 :  $(m_0, m_1) \leftarrow \mathcal{A}.\mathsf{P}((\text{seed}_{\mathbf{A}}, \mathbf{b}))$
7 :  $\mathbf{s}' \leftarrow\$ \beta_\mu(R_q^{l\times 1})$
8 :  $\mathbf{b}' \leftarrow \lfloor \mathbf{A}^T\cdot\mathbf{s}'+\mathbf{h} \rceil_{q\to p}$
9 :  $v' \leftarrow \lfloor \mathbf{b}^T\cdot\mathbf{s}' + h_1 \rceil_{q\to p}$
10 : $\hat{c} \leftarrow v' + (\lfloor m_u \rceil_{2\to p^2/q} \bmod p)$
11 : $u' \leftarrow \mathcal{A}.\mathsf{D}((\text{seed}_{\mathbf{A}}, \mathbf{b}), (\hat{c}, \mathbf{b}'))$
12 : $\textbf{return } (u' = u)$
```

$\text{Game}^4_{\mathcal{A}}$

```
1 :  $u \leftarrow\$ \mathcal{U}(\{0,1\})$
2 :  $\text{seed}_{\mathbf{A}} \leftarrow\$ \mathcal{U}(\{0,1\}^{256})$
3 :  $\mathbf{A} \leftarrow \mathsf{gen}(\text{seed}_{\mathbf{A}})$
4 :  Skip
5 :  $\mathbf{b} \leftarrow\$ \mathcal{U}(R_q^{l\times 1})$
6 :  $(m_0, m_1) \leftarrow \mathcal{A}.\mathsf{P}((\text{seed}_{\mathbf{A}}, \mathbf{b}))$
7 :  Skip
8 :  $\mathbf{b}' \leftarrow\$ \mathcal{U}(R_p^{l\times 1})$
9 :  $v' \leftarrow\$ \mathcal{U}(R_p)$
10 : $\hat{c} \leftarrow v' + (\lfloor m_u \rceil_{2\to p^2/q} \bmod p)$
11 : $u' \leftarrow \mathcal{A}.\mathsf{D}((\text{seed}_{\mathbf{A}}, \mathbf{b}), (\hat{c}, \mathbf{b}'))$
12 : $\textbf{return } (u' = u)$
```

Figure 3.8: Game Sequence in the Game-Playing Proof of Saber's PKE

```
$\text{Game}^{\text{GMLWR}}_{\mathcal{B}_0^{\mathcal{A}},l,\mu,q,p}(u)$
1 : $\text{seed}_{\mathbf{A}} \leftarrow\$ \; \mathcal{U}(\{0,1\}^{256})$
2 : $\mathbf{A} \leftarrow \mathsf{gen}(\text{seed}_{\mathbf{A}})$
3 : $\mathbf{s} \leftarrow\$ \; \beta_\mu(R_q^{l\times 1})$
4 : $\mathbf{b}_0 \leftarrow \lfloor \mathbf{A} \cdot \mathbf{s} + \mathbf{h} \rceil_{q\to p}$
5 : $\mathbf{b}_1 \leftarrow\$ \; \mathcal{U}(R_p^{l\times 1})$
6 : return
    $\mathcal{B}_0^{\mathcal{A}}(\text{seed}_{\mathbf{A}}, \mathbf{b}_u)$
    1 : $w \leftarrow\$ \; \mathcal{U}(\{0,1\})$
    2 : $\mathbf{A} \leftarrow \mathsf{gen}(\text{seed}_{\mathbf{A}})$
    3 : $(m_0, m_1) \leftarrow \mathcal{A}.\mathsf{P}((\text{seed}_{\mathbf{A}}, \mathbf{b}_u))$
    4 : $\mathbf{s}' \leftarrow\$ \; \beta_\mu(R_q^{l\times 1})$
    5 : $\mathbf{b}' \leftarrow \lfloor \mathbf{A}^T \cdot \mathbf{s}' + \mathbf{h} \rceil_{q\to p}$
    6 : $v' \leftarrow \mathbf{b}_u^T \cdot (\mathbf{s}' \bmod p) + (h_1 \bmod p)$
    7 : $\hat{c} \leftarrow \lfloor v' + \lfloor m_w \rceil_{2\to p} \rceil_{p\to 2\cdot t}$
    8 : $w' \leftarrow \mathcal{A}.\mathsf{D}((\text{seed}_{\mathbf{A}}, \mathbf{b}_u), (\hat{c}, \mathbf{b}'))$;
    9 : return $w' = w$;
```

Figure 3.9: Reduction from $\text{Game}^{\text{GMLWR}}_{\mathcal{B}_0^{\mathcal{A}},l,\mu,q,p}$ to Distinguishing $\text{Game}^0_{\mathcal{A}}$ and $\text{Game}^1_{\mathcal{A}}$

In this derivation, the third equality follows from the fact that $\mathcal{B}_0^{\mathcal{A}}(\text{seed}_{\mathbf{A}}, \mathbf{b}_u)$ perfectly simulates $\text{Game}^0_{\mathcal{A}}$ when $u = 0$ and $\text{Game}^1_{\mathcal{A}}$ when $u = 1$.

Step 2: $\text{Game}^1_{\mathcal{A}}$ **-** $\text{Game}^2_{\mathcal{A}}$ For the second step, we introduce a modification that results in an adversary against $\text{Game}^2_{\mathcal{A}}$ always acquiring at least as much information as an adversary against $\text{Game}^1_{\mathcal{A}}$. Consequently, given an adversary $\mathcal{A}$ against $\text{Game}^1_{\mathcal{A}}$, we can construct an adversary $\mathcal{R}^{\mathcal{A}}$ against $\text{Game}^2_{\mathcal{R}^{\mathcal{A}}}$ such that $\mathsf{Adv}^1(\mathcal{A}) = \mathsf{Adv}^2(\mathcal{R}^{\mathcal{A}})$. Specifically, assuming the $\hat{c}$ from $\text{Game}^2_{\mathcal{R}^{\mathcal{A}}}$ provides at least as much information as the $\hat{c}$ from $\text{Game}^1_{\mathcal{A}}$, the rationale behind this is the following. Foremost, because $\text{Game}^1_{\mathcal{A}}$ and $\text{Game}^2_{\mathcal{R}^{\mathcal{A}}}$ are identical in terms of their operations (apart from the computation of $\hat{c}$), adversary $\mathcal{A}$ against $\text{Game}^1_{\mathcal{A}}$ attempts to solve the exact same problem as adversary $\mathcal{R}^{\mathcal{A}}$ against $\text{Game}^2_{\mathcal{R}^{\mathcal{A}}}$, except that $\mathcal{R}^{\mathcal{A}}$ potentially has access to more information through the given $\hat{c}$. Following, if $\mathcal{R}^{\mathcal{A}}$ disposes of this additional information relative to $\text{Game}^1_{\mathcal{A}}$, calls $\mathcal{A}$ with the remaining information, and directly returns the values obtained from these calls to $\mathcal{A}$, $\mathcal{R}^{\mathcal{A}}$ achieves an advantage in $\text{Game}^2_{\mathcal{R}^{\mathcal{A}}}$ that is precisely equal to the advantage of $\mathcal{A}$ in $\text{Game}^1_{\mathcal{A}}$. Evidently, as with the previously discussed reductions from MLWR to GMLWR and XMLWR, this reasoning is only valid if $\mathcal{A}$ behaves identically between the reduction, i.e., when used as a sub-procedure by $\mathcal{R}^{\mathcal{A}}$ in $\text{Game}^2_{\mathcal{R}^{\mathcal{A}}}$, and a regular run of its own game, i.e., $\text{Game}^1_{\mathcal{A}}$. Since $\mathcal{A}$ is a black box and may be any feasible algorithm, this can only be guaranteed if $\mathcal{A}$ is unable to distinguish between these cases; concretely, this means that the adversary can not differentiate the information given in the reduction from the information given in a regular run of its own game. Figure 3.10 presents a reduction concerning $\text{Game}^1_{\mathcal{A}}$ and $\text{Game}^2_{\mathcal{R}^{\mathcal{A}}}$.

To prove that $\text{Game}^2_{\mathcal{A}}$ provides at least as much information as $\text{Game}^1_{\mathcal{A}}$ and the reduction in Figure 3.10 is correct, it suffices to show the two points below. Specifically, this is because the only difference between $\text{Game}^1_{\mathcal{A}}$ and $\text{Game}^2_{\mathcal{A}}$ regards the computation of $\hat{c}$.

1. The $\hat{c}$ given to the adversary in $\text{Game}^1_{\mathcal{A}}$ can always be computed from the $\hat{c}$ given to the adversary in $\text{Game}^2_{\mathcal{A}}$.

2. In the reduction of Figure 3.10, $\mathcal{R}^{\mathcal{A}}$ exactly computes (and calls $\mathcal{A}$ with) the $\hat{c}$ from $\text{Game}^1_{\mathcal{A}}$;

```
Game$^2_{\mathcal{R}^{\mathcal{A}}}$
1 : $u \leftarrow\$ \mathcal{U}(\{0,1\})$
2 : $\text{seed}_{\mathbf{A}} \leftarrow\$ \mathcal{U}(\{0,1\}^{256})$
3 : $\mathbf{A} \leftarrow \mathsf{gen}(\text{seed}_{\mathbf{A}})$
4 : Skip
5 : $\mathbf{b} \leftarrow\$ \mathcal{U}(R_p^{l\times 1})$
6 : $(m_0, m_1) \leftarrow$ $\mathcal{R}^{\mathcal{A}}.\mathsf{P}((\text{seed}_{\mathbf{A}}, \mathbf{b}))$
        1 : return $\mathcal{A}.\mathsf{P}((\text{seed}_{\mathbf{A}}, \mathbf{b}))$
7 : $\mathbf{s}' \leftarrow\$ \beta_\mu(R_q^{l\times 1})$
8 : $\mathbf{b}' \leftarrow \lfloor \mathbf{A}^T \cdot \mathbf{s}' + \mathbf{h} \rfloor_{q\to p}$
9 : $v' \leftarrow \mathbf{b}^T \cdot (\mathbf{s}' \bmod p) + (h_1 \bmod p)$
10 : $\hat{c} \leftarrow \lfloor v' + \lfloor m_u \rfloor_{2\to p} \rfloor_{p\to p^2/q}$
11 : $u' \leftarrow$ $\mathcal{R}^{\mathcal{A}}.\mathsf{D}((\text{seed}_{\mathbf{A}}, \mathbf{b}), (\hat{c}, \mathbf{b}'))$
        1 : $\hat{c}' \leftarrow \lfloor \hat{c} \rfloor_{p^2/q\to 2\cdot t}$
        2 : return $\mathcal{A}.\mathsf{D}((\text{seed}_{\mathbf{A}}, \mathbf{b}), (\hat{c}', \mathbf{b}'))$
```

Figure 3.10: Reduction from Game$^2_{\mathcal{R}^{\mathcal{A}}}$ to Game$^1_{\mathcal{A}}$

in particular, $\mathcal{R}^{\mathcal{A}}$ does so based on the $\hat{c}$ provided in Game$^2_{\mathcal{R}^{\mathcal{A}}}$.

Certainly, both of these points are guaranteed by the $\frac{q}{p} \leq \frac{p}{2t}$ assumption from the security theorem, the construction of the games, and the construction of the reduction.

Commencing with the analysis of the two above-mentioned points, consider $x = v' + \lfloor m_u \rfloor_{2\to p}$, where v' and m_u are as in Game$^1_{\mathcal{A}}$ and Game$^2_{\mathcal{A}}$. Then, because $v' + \lfloor m_u \rfloor_{2\to p}$ is an element of R_p, x is as well; notably, this implies that each coefficient of x is an element of $\mathbb{Z}_p$. Denoting the binary representation of a coefficient of x by $a_{\epsilon_p-1} \dots a_0$, the derivation below particularizes the modular scaling and flooring operation performed on x in Game$^1_{\mathcal{A}}$. Here, we utilize the assumption $\epsilon_t + 1 < \epsilon_p$ to infer that this operation effectively performs a right bit-shift.

$$\begin{aligned}\lfloor a_{\epsilon_p-1} \dots a_0 \rfloor_{p\to 2\cdot t} &= a_{\epsilon_p-1} \dots a_{\epsilon_p-(\epsilon_t+1)} \\ &= a_{\epsilon_p-1} \dots a_{\epsilon_p-\epsilon_t-1}\end{aligned}$$

Thus, for each coefficient $a_{\epsilon_p-1} \dots a_0$ of x, the corresponding coefficient of $\hat{c}$ equals $a_{\epsilon_p-1} \dots a_{\epsilon_p-\epsilon_t-1}$.

In contrast to Game$^1_{\mathcal{A}}$, Game$^2_{\mathcal{A}}$ constructs $\hat{c}$ by applying a modular scaling and flooring operation with target modulus $\frac{p^2}{q}$; as such, for each coefficient $a_{\epsilon_p-1} \dots a_0$ of x, the modular scaling and flooring operation in Game$^2_{\mathcal{A}}$ computes the corresponding coefficient of $\hat{c}$ as follows.

$$\begin{aligned}\lfloor a_{\epsilon_p-1} \dots a_0 \rfloor_{p\to p^2/q} &= a_{\epsilon_p-1} \dots a_{\epsilon_p-(2\cdot\epsilon_p-\epsilon_q)} \\ &= a_{\epsilon_p-1} \dots a_{\epsilon_q-\epsilon_p}\end{aligned}$$

In this derivation, the first equality holds due to the fact that $p = 2^{\epsilon_p}$ and $\frac{p^2}{q} = 2^{2\cdot\epsilon_p-\epsilon_q}$. Namely, these imply that $\frac{p^2}{q} < p$ if and only if $2\cdot\epsilon_p - \epsilon_q < \epsilon_p$; in turn, this latter inequality follows from $\epsilon_p < \epsilon_q$, one of Saber's parameter requirements (see Section 3.1). In consequence, the considered modular scaling and flooring operation is equivalent to a right bit-shift of $\epsilon_p - (2\cdot\epsilon_p - \epsilon_q) = \epsilon_q - \epsilon_p$ bits.

Comparing the corresponding coefficients of $\hat{c}$ between the two games, we see that they are equal in their *most* significant bits; however, depending on the value of $\epsilon_p - \epsilon_t - 1$ and $\epsilon_q - \epsilon_p$, they may differ in their *least* significant bits. Specifically, if $\epsilon_q - \epsilon_p \leq \epsilon_p - \epsilon_t - 1$, the coefficients of the $\hat{c}$ from $\text{Game}^2_{\mathcal{A}}$ contain at least the same bits as their counterparts from $\text{Game}^1_{\mathcal{A}}$; hence, provided this inequality holds, the $\hat{c}$ from $\text{Game}^2_{\mathcal{A}}$ would provide the adversary with at least as much information as the $\hat{c}$ from $\text{Game}^1_{\mathcal{A}}$. Employing the assumption stipulated by the security theorem, i.e., $\frac{q}{p} \leq \frac{p}{2 \cdot t}$, the ensuing derivation confirms that this indeed the case.

$$
\begin{aligned}
\frac{q}{p} \leq \frac{p}{2 \cdot t} &\Leftrightarrow \frac{2^{\epsilon_q}}{2^{\epsilon_p}} \leq \frac{2^{\epsilon_p}}{2^{\epsilon_t + 1}} \\
&\Leftrightarrow 2^{\epsilon_q - \epsilon_p} \leq 2^{\epsilon_p - (\epsilon_t + 1)} \\
&\Leftrightarrow \epsilon_q - \epsilon_p \leq \epsilon_p - (\epsilon_t + 1) \\
&\Leftrightarrow \epsilon_q - \epsilon_p \leq \epsilon_p - \epsilon_t - 1
\end{aligned}
$$

As such, $\text{Game}^2_{\mathcal{A}}$ provides at least as much information to the adversary as $\text{Game}^1_{\mathcal{A}}$; moreover, given the information from $\text{Game}^2_{\mathcal{A}}$, we can exactly replicate the corresponding information of $\text{Game}^1_{\mathcal{A}}$. More precisely, because $\epsilon_q - \epsilon_p \leq \epsilon_p - \epsilon_t - 1$, each coefficient a of the $\hat{c}$ from $\text{Game}^2_{\mathcal{A}}$ is constructed as follows.

$$a = a_{\epsilon_p - 1} \dots a_{\epsilon_p - \epsilon_t - 1} \dots a_{\epsilon_q - \epsilon_p}$$

Naturally, in case $\epsilon_q - \epsilon_p = \epsilon_p - \epsilon_t - 1$, this reduces to $a = a_{\epsilon_p - 1} \dots a_{\epsilon_p - \epsilon_t - 1}$.

In consequence of the above, replicating the $\hat{c}$ from $\text{Game}^1_{\mathcal{A}}$ merely requires, for each coefficient, discarding the bits that are *less* significant than $a_{\epsilon_p - \epsilon_t - 1}$; indeed, this is exactly what transpires in the reduction of Figure 3.10. Subsequently, the result, $\hat{c}'$, is given as input to the adversary against $\text{Game}^1_{\mathcal{A}}$. The imminent derivation proves that this is indeed the case.

$$
\begin{aligned}
\lfloor a_{\epsilon_p - 1} \dots a_{\epsilon_p - \epsilon_t - 1} \dots a_{\epsilon_q - \epsilon_p} \rfloor_{p^2/q \to 2 \cdot t} &= a_{\epsilon_p - 1} \dots a_{(\epsilon_q - \epsilon_p) + ((2 \cdot \epsilon_p - \epsilon_q) - (\epsilon_t + 1))} \\
&= a_{\epsilon_p - 1} \dots a_{\epsilon_p - \epsilon_t - 1}
\end{aligned}
$$

As before, the modular scaling and flooring operator effectively acts as a right bit-shift; nevertheless, here, the number of shifted bits might equal 0. Namely, if $\epsilon_q - \epsilon_p = \epsilon_p - \epsilon_t - 1$, then the argument to the operator reduces to $a_{\epsilon_p - 1} \dots a_{\epsilon_p - \epsilon_t - 1}$. Certainly, this argument already equals the final result, implying that the operator's application has no effect; in other words, the operator performs a bit-shift of 0 bits.

Having shown that the two points provided at the beginning of this step are satisfied by the considered games and reduction, we conclude that $\text{Game}^2_{\mathcal{A}}$ indeed provides its adversary with at least as much information as $\text{Game}^1_{\mathcal{A}}$ does. Furthermore, the reduction depicted in Figure 3.10 is correct; hence, for all adversaries $\mathcal{A}$ against $\text{Game}^1_{\mathcal{A}}$, there exists an adversary $\mathcal{R}^{\mathcal{A}}$ against $\text{Game}^2_{\mathcal{R}^{\mathcal{A}}}$ such that $\Pr\left[\text{Game}^1_{\mathcal{A}} = 1\right] = \Pr\left[\text{Game}^2_{\mathcal{R}^{\mathcal{A}}} = 1\right]$. Utilizing this latter fact, we can trivially derive the desired result as follows.

$$
\begin{aligned}
\forall_{\mathcal{A}} \exists_{\mathcal{R}^{\mathcal{A}}} :\ & \Pr\left[\text{Game}^1_{\mathcal{A}} = 1\right] = \Pr\left[\text{Game}^2_{\mathcal{R}^{\mathcal{A}}} = 1\right] \\
& \Rightarrow \\
\forall_{\mathcal{A}} \exists_{\mathcal{R}^{\mathcal{A}}} :\ & \left| \Pr\left[\text{Game}^1_{\mathcal{A}} = 1\right] - \frac{1}{2} \right| = \left| \Pr\left[\text{Game}^2_{\mathcal{R}^{\mathcal{A}}} = 1\right] - \frac{1}{2} \right| \\
& \Rightarrow \\
\forall_{\mathcal{A}} \exists_{\mathcal{R}^{\mathcal{A}}} :\ & \mathsf{Adv}^1(\mathcal{A}) = \mathsf{Adv}^2(\mathcal{R}^{\mathcal{A}})
\end{aligned}
$$

Step 3: $\text{Game}^2_{\mathcal{A}}$ **-** $\text{Game}^3_{\mathcal{A}}$ In this step, similarly to the preceding step, we exclusively introduce alterations that provide an adversary against $\text{Game}^3_{\mathcal{A}}$ with at least as much information as an adversary against $\text{Game}^2_{\mathcal{A}}$. Therefore, given any adversary $\mathcal{A}$ against $\text{Game}^2_{\mathcal{A}}$, we can construct an adversary $\mathcal{R}^{\mathcal{A}}$ against $\text{Game}^3_{\mathcal{R}^{\mathcal{A}}}$ such that $\mathsf{Adv}^2(\mathcal{A}) = \mathsf{Adv}^3(\mathcal{R}^{\mathcal{A}})$. Figure 3.11 provides such a reduction.

```
$\text{Game}^3_{\mathcal{R}^{\mathcal{A}}}$
1 : $u \leftarrow\$ \mathcal{U}(\{0,1\})$
2 : $\text{seed}_{\mathbf{A}} \leftarrow\$ \mathcal{U}(\{0,1\}^{256})$
3 : $\mathbf{A} \leftarrow \mathsf{gen}(\text{seed}_{\mathbf{A}})$
4 : Skip
5 : $\mathbf{b} \leftarrow\$ \mathcal{U}(R_q^{l\times 1})$
6 : $(m_0, m_1) \leftarrow$ $\mathcal{R}^{\mathcal{A}}.\mathsf{P}((\text{seed}_{\mathbf{A}}, \mathbf{b}))$
        1 : $\mathbf{b}_p \leftarrow \mathbf{b} \bmod p$
        2 : return $\mathcal{A}.\mathsf{P}((\text{seed}_{\mathbf{A}}, \mathbf{b}_p))$
7 : $\mathbf{s}' \leftarrow\$ \beta_\mu(R_q^{l\times 1})$
8 : $\mathbf{b}' \leftarrow \lfloor \mathbf{A}^T \cdot \mathbf{s}' + \mathbf{h} \rceil_{q\to p}$
9 : $v' \leftarrow \lfloor \mathbf{b}^T \cdot \mathbf{s}' + h_1 \rceil_{q\to p}$
10 : $\hat{c} \leftarrow v' + (\lfloor m_u \rceil_{2\to p^2/q} \bmod p)$
11 : return $\mathcal{R}^{\mathcal{A}}.\mathsf{D}((\text{seed}_{\mathbf{A}}, \mathbf{b}), (\hat{c}, \mathbf{b}'))$
        1 : $\mathbf{b}_p \leftarrow \mathbf{b} \bmod p$
        2 : $\hat{c}' \leftarrow \hat{c} \bmod p^2/q$
        3 : return $\mathcal{A}.\mathsf{P}((\text{seed}_{\mathbf{A}}, \mathbf{b}_p), (\hat{c}', \mathbf{b}'))$;
```

Figure 3.11: Reduction from $\text{Game}^3_{\mathcal{R}^{\mathcal{A}}}$ to $\text{Game}^2_{\mathcal{A}}$

To show that $\text{Game}^3_{\mathcal{A}}$ provides at least as much information as $\text{Game}^2_{\mathcal{A}}$ and the reduction presented in Figure 3.11 is correct, we argue the following two points.

1. Sampling $\mathbf{b}$ from $\mathcal{U}(R_q^{l\times 1})$ and, subsequently, reducing it modulo p is well-defined; moreover, this process is equivalent to sampling $\mathbf{b}$ directly from $\mathcal{U}(R_p^{l\times 1})$. Here, "equivalent" refers to the fact that both processes produce identically distributed outcomes.

2. The $\hat{c}$ from $\text{Game}^3_{\mathcal{A}}$ provides at least as much information as the $\hat{c}$ from $\text{Game}^2_{\mathcal{A}}$. Furthermore, in the reduction of Figure 3.11, $\mathcal{R}^{\mathcal{A}}$ precisely computes (and calls $\mathcal{A}$ with) the $\hat{c}$ from $\text{Game}^2_{\mathcal{A}}$; particularly, $\mathcal{R}^{\mathcal{A}}$ does so based on the $\hat{c}$ given in $\text{Game}^3_{\mathcal{R}^{\mathcal{A}}}$. Crucially, the computed value must be consistent with the other artifacts that $\mathcal{R}^{\mathcal{A}}$ provides to $\mathcal{A}$; that is, the combination of values that $\mathcal{R}^{\mathcal{A}}$ provides as input to $\mathcal{A}$ must constitute a valid run of $\text{Game}^2_{\mathcal{A}}$.

The rationale behind proving these points is analogous to before. That is, we can only expect an adversary to behave identically between a reduction and a regular run of its own game if it cannot distinguish between these cases. Alternatively stated, the information given to the adversary in a reduction and a regular run of its own game must be indistinguishable to this adversary. The above two points are necessary and sufficient to show this for the considered games and reduction; in particular, this is because $\mathbf{b}$ and $\hat{c}$ are the only artifacts given to the adversary that differ between these two games.

Regarding the first point, consider $\mathbf{b}$ from $\text{Game}^3_{\mathcal{A}}$; this vector is sampled uniformly at random from $R_q^{l\times 1}$. As such, since all coefficients of the polynomials in this vector are elements of $\mathbb{Z}_q$ and $p \mid q$, reduction modulo p is well-defined for these coefficients. In turn, the extension of this modular reduction to the complete vector, i.e., $\mathbf{b} \bmod p$, is a well-defined operation as well.

Furthermore, the resulting vector, i.e., $\mathbf{b}_p$ in Figure 3.11, is an element of $R_p^{l\times 1}$.

Next, deriving the uniformity of $\mathbf{b}_p$ over $R_p^{l\times 1}$ involves the ensuing observations. First, each element of $R_q^{l\times 1}$ comprises l polynomials of which each has n coefficients. Following, because each coefficient can be any of the q elements in $\mathbb{Z}_q$, the total number of elements in $R_q^{l\times 1}$ equals $(q^n)^l = q^{n\cdot l}$. As a result, the probability of sampling any particular element of $R_q^{l\times 1}$ uniformly at random is $\frac{1}{q^{n\cdot l}}$; that is, if $\mathbf{x}$ is a fixed element of $R_q^{l\times 1}$ and $\mathbf{b} \leftarrow\$ \; \mathcal{U}(R_q^{l\times 1})$, then $\Pr[\mathbf{b} = \mathbf{x}] = \frac{1}{q^{n\cdot l}}$. A similar line of reasoning applies to the probability of sampling an element of $R_p^{l\times 1}$ uniformly at random; specifically, if $\mathbf{y}$ is a fixed element of $R_p^{l\times 1}$ and $\mathbf{d} \leftarrow\$ \; \mathcal{U}(R_p^{l\times 1})$, then $\Pr[\mathbf{d} = \mathbf{y}] = \frac{1}{p^{n\cdot l}}$. Second, since $p = 2^{\epsilon_p}$, reducing an integer modulo p is equivalent to setting all bits *more* significant than the ϵ_p-th bit to 0. Moreover, as $q = 2^{\epsilon_q}$, any element of $\mathbb{Z}_q$ and $\mathbb{Z}_p$ can be represented by ϵ_q and ϵ_p bits, respectively. Therefore, reducing an element of $\mathbb{Z}_q$ modulo p is equivalent to setting the $\epsilon_q - \epsilon_p$ most significant bits of this element to 0. Certainly, this implies that for any element $a \in \mathbb{Z}_p$, a total of $2^{\epsilon_q - \epsilon_p} = \frac{q}{p}$ elements of $\mathbb{Z}_q$ map to a when reduced modulo p. Applying this reasoning to each coefficient of every polynomial in $\mathbf{b}$, we deduce that the number of elements in $R_q^{l\times 1}$ that, when reduced modulo p, map to a particular element in $R_p^{l\times 1}$ equals $\frac{q^{n\cdot l}}{p^{n\cdot l}}$. Lastly, combining the above observations, we conclude that if $\mathbf{b}$ is sampled from $\mathcal{U}(R_q^{l\times 1})$, then $\mathbf{b}_p = \mathbf{b} \bmod p$ is uniformly distributed over $R_p^{l\times 1}$. More formally, let $\mathbf{y}$ be a fixed element of $R_p^{l\times 1}$, $X = \{\mathbf{x} \mid \mathbf{x} \in R_q^{l\times 1} \wedge \mathbf{x} \bmod p = \mathbf{y}\} = \{\mathbf{x}_1, \ldots, \mathbf{x}_{\frac{q^{n\cdot l}}{p^{n\cdot l}}}\}$ the set of all elements in $R_q^{l\times 1}$ that equal $\mathbf{y}$ after reduction modulo p, $\mathbf{b} \leftarrow\$ \; \mathcal{U}(R_q^{l\times 1})$, and $\mathbf{d} \leftarrow\$ \; \mathcal{U}(R_p^{l\times 1})$; then, the following holds.

$$\begin{aligned}
\Pr[\mathbf{b} \bmod p = \mathbf{y}] &= \Pr[\mathbf{b} \in X] \\
&= \Pr\left[\mathbf{b} = \mathbf{x}_1 \vee \ldots \vee \mathbf{b} = \mathbf{x}_{\frac{q^{n\cdot l}}{p^{n\cdot l}}}\right] \\
&= \Pr[\mathbf{b} = \mathbf{x}_1] + \ldots + \Pr\left[\mathbf{b} = \mathbf{x}_{\frac{q^{n\cdot l}}{p^{n\cdot l}}}\right] \\
&= \frac{1}{q^{n\cdot l}} + \ldots + \frac{1}{q^{n\cdot l}} \\
&= \frac{1}{q^{n\cdot l}} \cdot \frac{q^{n\cdot l}}{p^{n\cdot l}} \\
&= \frac{1}{p^{n\cdot l}} \\
&= \Pr[\mathbf{d} = \mathbf{y}]
\end{aligned}$$

Concerning the second point, consider the computation of $\mathbf{b}^T \cdot \mathbf{s}' + h_1$ in $\text{Game}^3_{\mathcal{A}}$, with $\mathbf{b}, \mathbf{s}' \in R_q^{l\times 1}$ and $h_1 \in R_q$. Since its result is an element of R_q, this computation may equivalently be denoted as $(\mathbf{b}^T \cdot \mathbf{s}' + h_1) \bmod q$. Employing this equivalent notation, the ensuing derivation shows that additionally reducing this computation modulo p produces an identical outcome to carrying out the same computation with each artifact individually reduced modulo p (in addition to reducing the result modulo p). Here, the initial equality follows from the fact that $p \mid q$.

$$\begin{aligned}
((\mathbf{b}^T \cdot \mathbf{s}' + h_1) \bmod q) \bmod p &= ((\mathbf{b}^T \cdot \mathbf{s}' + h_1) \bmod p \\
&= ((\mathbf{b}^T \bmod p) \cdot (\mathbf{s}' \bmod p) + (h_1 \bmod p)) \bmod p
\end{aligned}$$

Certainly, this implies that the reduction modulo p of $\mathbf{b}^T \cdot \mathbf{s}' + h_1$ in $\text{Game}^3_{\mathcal{A}}$ would lead to the same (distribution of the) result as the computation that $\text{Game}^2_{\mathcal{A}}$ carries out and assigns to v'. Namely, the sole difference between these computations concerns the manner in which $\mathbf{b}$ is obtained. Nevertheless, as can be extracted from the above derivation, the reduction modulo p of $\mathbf{b}^T \cdot \mathbf{s}' + h_1$ in $\text{Game}^3_{\mathcal{A}}$ effectively leads to the reduction modulo p of $\mathbf{b}$. As per the previous point, this reduced $\mathbf{b}$, i.e., an element originally sampled from $\mathcal{U}(R_q^{l\times 1})$ and then reduced modulo

p, is well-defined and identically distributed to the $\mathbf{b}$ utilized in $\text{Game}^2_{\mathcal{A}}$, viz., uniformly over $R_p^{l\times 1}$. Consequently, in essence, reducing the result of $\mathbf{b}^T \cdot \mathbf{s}' + h_1$ in $\text{Game}^3_{\mathcal{A}}$ modulo p would produce the same result as the analogous computation in $\text{Game}^2_{\mathcal{A}}$; equivalently, the ϵ_p least significant bits of the corresponding coefficients of $\mathbf{b}^T \cdot \mathbf{s}' + h_1$ are identical between the two games. In consequence of this equality, if $a_{\epsilon_q-1} \dots a_{\epsilon_p-1} \dots a_0$ denotes the binary representation of a coefficient of $\mathbf{b}^T \cdot \mathbf{s}' + h_1$ in $\text{Game}^3_{\mathcal{A}}$, then $a_{\epsilon_p-1} \dots a_0$ denotes the binary representation of the corresponding coefficient of the analogous computation in $\text{Game}^2_{\mathcal{A}}$. Although $\text{Game}^2_{\mathcal{A}}$ immediately assigns the result of this computation to v', $\text{Game}^3_{\mathcal{A}}$ precedes this assignment by modular scaling and flooring from modulo q to modulo p. That is, for each coefficient, $\text{Game}^3_{\mathcal{A}}$ computes $\lfloor a_{\epsilon_q-1} \dots a_{\epsilon_p-1} \dots a_0 \rceil_{q\to p} = a_{\epsilon_q-1} \dots a_{\epsilon_p-1} \dots a_{\epsilon_q-\epsilon_p}$, the result of which it assigns to v'. Comparing the values of v' between the two games, we see that the $\epsilon_p - (\epsilon_q - \epsilon_p) = 2 \cdot \epsilon_p - \epsilon_q$ *least* significant bits of the coefficients of v' from $\text{Game}^3_{\mathcal{A}}$ are precisely equal to the same number of *most* significant bits of their counterparts from $\text{Game}^2_{\mathcal{A}}$.

After the computation of v', both games compute $\hat{c}$ by adding a scaled version of m_u to v' and, in the case of $\text{Game}^2_{\mathcal{A}}$, subsequently modular scaling and flooring the result. More precisely, in $\text{Game}^2_{\mathcal{A}}$, $\hat{c}$ is computed as $\lfloor v' + \lfloor m_u \rceil_{2\to p} \rceil_{p \to p^2/q}$. The following derivation unfolds this operation for arbitrary (corresponding) coefficients of v' and m_u, denoted by their respective binary representations $a_{\epsilon_p-1} \dots a_0$ (as above) and b_0.

$$\begin{aligned}
\lfloor a_{\epsilon_p-1} \dots a_0 + \lfloor b_0 \rceil_{2\to p} \rceil_{p\to p^2/q} &= \lfloor a_{\epsilon_p-1} \dots a_0 + b_0 0^{\epsilon_p-1} \rceil_{p\to p^2/q} \\
&= \lfloor (a_{\epsilon_p-1} + b_0) \dots a_0 \rceil_{p\to p^2/q} \\
&= (a_{\epsilon_p-1} + b_0) \dots a_{\epsilon_q-\epsilon_p}
\end{aligned}$$

In this derivation, $(a_{\epsilon_p-1} + b_0)$ represents the (single) bit value resulting from the addition of the a_{ϵ_p-1} and b_0 bits (modulo 2). Nevertheless, because the addition of $a_{\epsilon_p-1} \dots a_0$ and $b_0 0^{\epsilon_p-1}$ is performed in $\mathbb{Z}_p$, it is not possible for the addition of the a_{ϵ_p-1} and b_0 bits to create an $(\epsilon_p + 1)$-th bit through their potential carry; namely, since $\mathbb{Z}_p$ implicitly induces a reduction modulo $p = 2^{\epsilon_p}$, any bits *more* significant than the ϵ_p-th bit are effectively discarded.

Concerning $\text{Game}^3_{\mathcal{A}}$, $\hat{c}$ is computed as $v' + (\lfloor m_u \rceil_{2\to p^2/q} \bmod p)$. In this computation, the reduction modulo p is merely used to accentuate the interpretation of $\lfloor m_u \rceil_{2\to p^2/q}$ as an element of R_p; that is, instead of an element of $R_{p^2/q}$, which is the range of the $\lfloor \cdot \rceil_{2\to p^2/q}$ operator. Certainly, since $\frac{p^2}{q} < p$, this modular reduction cannot affect the value of $\lfloor m_u \rceil_{2\to p^2/q}$. The ensuing derivation serves a similar purpose to the analogous derivation for the $\hat{c}$ of $\text{Game}^2_{\mathcal{A}}$; moreover, it utilizes the same denotation for the binary representation of a coefficient of m_u. However, to emphasize the relation between the v' values from the different games, the subsequent derivation uses $a_{\epsilon_q-1} \dots a_{\epsilon_p-1} \dots a_{\epsilon_q-\epsilon_p}$ to denote a coefficient of v'. Indeed, this representation is consistent with the preceding computations, indicating the value of such a coefficient relative to its counterpart from $\text{Game}^2_{\mathcal{A}}$. As a consequence, based on the utilized representations, we can directly deduce which bits are equal by their matching index.

$$\begin{aligned}
a_{\epsilon_q-1} \dots a_{\epsilon_p-1} \dots a_{\epsilon_q-\epsilon_p} + (\lfloor b_0 \rceil_{2\to p^2/q} \bmod p) &= a_{\epsilon_q-1} \dots a_{\epsilon_p-1} \dots a_{\epsilon_q-\epsilon_p} + (b_0 0^{2\cdot\epsilon_p-\epsilon_q-1} \bmod p) \\
&= a_{\epsilon_q-1} \dots a_{\epsilon_p-1} \dots a_{\epsilon_q-\epsilon_p} + 0^{\epsilon_q-\epsilon_p} b_0 0^{2\cdot\epsilon_p-\epsilon_q-1} \\
&= d_{\epsilon_q-1} \dots d_{\epsilon_p} (a_{\epsilon_p-1} + b_0) a_{\epsilon_p-2} \dots a_{\epsilon_q-\epsilon_p}
\end{aligned}$$

Here, $(a_{\epsilon_p-1} + b_0)$ is used in the same manner as in the preceding derivation and each d_i represents a bit that might be influenced by potential carries. Moreover, the last equality follows from the fact that both a_{ϵ_p-1} and b_0 are exactly the $(2 \cdot \epsilon_p - \epsilon_q)$-th bit of their respective values. In particular, for a_{ϵ_p-1}, this is implied by $\epsilon_p - (\epsilon_q - \epsilon_p) = 2 \cdot \epsilon_p - \epsilon_q$; for b_0, this is a direct consequence of the $\lfloor \cdot \rceil_{2\to p^2/q}$ operator. Furthermore, as before, because the final addition is carried out in $\mathbb{Z}_p$, any potential $(\epsilon_p + 1)$-th bit resulting from the addition is effectively discarded through the implicit reduction modulo p. Lastly, comparing the outcomes of the previous two derivations, we see that the $2 \cdot \epsilon_p - \epsilon_q$ least significant bits of the (coefficients of the) $\hat{c}$ from $\text{Game}^3_{\mathcal{A}}$ are identical to their analogs from $\text{Game}^2_{\mathcal{A}}$.

As a result of the above, the $\hat{c}$ from $\text{Game}^3_{\mathcal{A}}$ provides at least as much information as its counterpart from $\text{Game}^2_{\mathcal{A}}$. Additionally, the latter can be constructed by discarding the $\epsilon_q - \epsilon_p$ most significant bits of the coefficients of the former; certainly, this is exactly what happens in the reduction of Figure 3.11. Specifically, given the $\hat{c}$ from $\text{Game}^3_{\mathcal{R}^{\mathcal{A}}}$, $\mathcal{R}^{\mathcal{A}}$ constructs $\hat{c}'$ by computing $\hat{c} \bmod \frac{p^2}{q} = \hat{c} \bmod 2^{2 \cdot \epsilon_p - \epsilon_q}$. Since this modular reduction effectively discards all bits that are *more* significant than the $(2 \cdot \epsilon_p - \epsilon_q)$-th bit, this produces the corresponding $\hat{c}$ from $\text{Game}^2_{\mathcal{A}}$. In particular, as can be extracted from the preceding discussion, this $\hat{c}$ would result from the specific run of $\text{Game}^2_{\mathcal{A}}$ in which $\mathbf{b}$ equals the $\mathbf{b}_p$ computed (and provided to $\mathcal{A}$) in the reduction; indeed, this implies that the combination of values provided to $\mathcal{A}$ by $\mathcal{R}^{\mathcal{A}}$ constitute a valid run of $\text{Game}^2_{\mathcal{A}}$.

Finally, having shown that the requirements specified at the beginning of this step hold, we conclude that $\text{Game}^3_{\mathcal{A}}$ provides at least as much information to its adversary as $\text{Game}^2_{\mathcal{A}}$ does. Moreover, the reduction depicted in Figure 3.11 is correct and, thus, for any adversary $\mathcal{A}$ against $\text{Game}^2_{\mathcal{A}}$, there exists an adversary $\mathcal{R}^{\mathcal{A}}$ against $\text{Game}^3_{\mathcal{R}^{\mathcal{A}}}$ such that $\Pr\left[\text{Game}^2_{\mathcal{A}} = 1\right] = \Pr\left[\text{Game}^3_{\mathcal{R}^{\mathcal{A}}} = 1\right]$. At last, the desired result can be deduced through the derivation below.

$$\begin{aligned}
\forall_{\mathcal{A}} \exists_{\mathcal{R}^{\mathcal{A}}} : \;& \Pr\left[\text{Game}^2_{\mathcal{A}} = 1\right] = \Pr\left[\text{Game}^3_{\mathcal{R}^{\mathcal{A}}} = 1\right] \\
& \Rightarrow \\
\forall_{\mathcal{A}} \exists_{\mathcal{R}^{\mathcal{A}}} : \;& \left|\Pr\left[\text{Game}^2_{\mathcal{A}} = 1\right] - \frac{1}{2}\right| = \left|\Pr\left[\text{Game}^3_{\mathcal{R}^{\mathcal{A}}} = 1\right] - \frac{1}{2}\right| \\
& \Rightarrow \\
\forall_{\mathcal{A}} \exists_{\mathcal{R}^{\mathcal{A}}} : \;& \mathsf{Adv}^2(\mathcal{A}) = \mathsf{Adv}^3(\mathcal{R}^{\mathcal{A}})
\end{aligned}$$

Step 4: $\text{Game}^3_{\mathcal{A}}$ **-** $\text{Game}^4_{\mathcal{A}}$ In the final step, we change the manner in which $\mathbf{b}'$ and v' are obtained. Namely, instead of computing these values by $\lfloor \mathbf{A}^T \cdot \mathbf{s}' + \mathbf{h} \rceil_{q \to p}$ and $\lfloor \mathbf{b}^T \cdot \mathbf{s}' + h_1 \rceil_{q \to p}$, as is done in $\text{Game}^3_{\mathcal{A}}$, they are sampled uniformly at random from their respective domains in $\text{Game}^4_{\mathcal{A}}$. As a consequence of this adjustment, $\mathbf{s}'$ becomes redundant and, hence, is removed from $\text{Game}^4_{\mathcal{A}}$.

In $\text{Game}^3_{\mathcal{A}}$, the pair $(\mathbf{A}, \mathbf{b}')$ comprises l GMLWR samples with respect to secret $\mathbf{s}'$, and the pair $(\mathbf{b}, v')$ constitutes a single MLWR sample for the same secret; contrarily, in $\text{Game}^4_{\mathcal{A}}$, these two pairs are sampled uniformly at random from their respective domains. As such, any adversary $\mathcal{A}$ distinguishing between $\text{Game}^3_{\mathcal{A}}$ and $\text{Game}^4_{\mathcal{A}}$ can be used to construct an adversary against the corresponding XMLWR game. Figure 3.12 contains such a reduction. Similarly to the reduction in the first step, the $\lfloor \cdot \rceil_{q \to p}$ operations from the XMLWR game's specification are replaced by the equivalent modular scaling and flooring operations under the addition of a constant exclusively consisting of (entries with only) $2^{\epsilon_q - \epsilon_p - 1}$ coefficients.

With regards to the reduction in Figure 3.12, we derive a result analogous to the result of the first step. That is, for any adversary $\mathcal{A}$ distinguishing between $\text{Game}^3_{\mathcal{A}}$ and $\text{Game}^4_{\mathcal{A}}$, there exists an adversary $\mathcal{B}^{\mathcal{A}}_1$ against the corresponding instance of the XMLWR game such that $\left|\Pr\left[\text{Game}^3_{\mathcal{A}} = 1\right] - \Pr\left[\text{Game}^4_{\mathcal{A}} = 1\right]\right| = \mathsf{Adv}^{\text{XMLWR}}_{l,\mu,q,p}(\mathcal{B}^{\mathcal{A}}_1)$. Specifically, following an identical line of reasoning to that of the first step, we deduce this result as follows.

$$\begin{aligned}
\forall_{\mathcal{A}} \exists_{\mathcal{B}^{\mathcal{A}}_1} : \mathsf{Adv}^{\text{XMLWR}}_{l,\mu,q,p}(\mathcal{B}^{\mathcal{A}}_1) &= \left|\Pr\left[\text{Game}^{\text{XMLWR}}_{\mathcal{B}^{\mathcal{A}}_1,l,\mu,q,p}(1) = 1\right] - \Pr\left[\text{Game}^{\text{XMLWR}}_{\mathcal{B}^{\mathcal{A}}_1,l,\mu,q,p}(0) = 1\right]\right| \\
&= \left|\Pr[w' = w \mid u = 1] - \Pr[w' = w \mid u = 0]\right| \\
&= \left|\Pr\left[\text{Game}^4_{\mathcal{A}} = 1\right] - \Pr\left[\text{Game}^3_{\mathcal{A}} = 1\right]\right| \\
&= \left|\Pr\left[\text{Game}^3_{\mathcal{A}} = 1\right] - \Pr\left[\text{Game}^4_{\mathcal{A}} = 1\right]\right|
\end{aligned}$$

```
$\text{Game}^{\text{XMLWR}}_{\mathcal{B}_1^{\mathcal{A}},l,\mu,q,p}(u)$
1 : $\text{seed}_{\mathbf{A}} \leftarrow\$ \mathcal{U}(\{0,1\}^{256})$
2 : $\mathbf{A} \leftarrow \mathsf{gen}(\text{seed}_{\mathbf{A}})$
3 : $\mathbf{s} \leftarrow\$ \beta_\mu(R_q^{l\times 1})$
4 : $\mathbf{b}_0 \leftarrow \lfloor \mathbf{A}^T \cdot \mathbf{s} + \mathbf{h} \rceil_{q\to p}$
5 : $\mathbf{b}_1 \leftarrow\$ \mathcal{U}(R_p^{l\times 1})$
6 : $\mathbf{a} \leftarrow\$ \mathcal{U}(R_q^{1\times l})$
7 : $d_0 \leftarrow \lfloor \mathbf{a} \cdot \mathbf{s} + h_1 \rceil_{q\to p}$
8 : $d_1 \leftarrow\$ \mathcal{U}(R_p)$
9 : return $\mathcal{B}_1^{\mathcal{A}}(\text{seed}_{\mathbf{A}}, \mathbf{b}_u, \mathbf{a}, d_u)$
        1 : $w \leftarrow\$ \mathcal{U}(\{0,1\})$;
        2 : $(m_0, m_1) \leftarrow \mathcal{A}.\mathsf{P}((\text{seed}_{\mathbf{A}}, \mathbf{a}^T))$;
        3 : $\hat{c} \leftarrow d_u + (\lfloor m_w \rceil_{2\to p^2/q} \bmod p)$
        4 : $w' \leftarrow \mathcal{A}.\mathsf{D}((\text{seed}_{\mathbf{A}}, \mathbf{a}^T), (\hat{c}, \mathbf{b}_u))$;
        5 : return $w = w'$;
```

Figure 3.12: Reduction from $\text{Game}^{\text{XMLWR}}_{\mathcal{B}_1^{\mathcal{A}},l,\mu,q,p}$ to Distinguishing $\text{Game}^3_{\mathcal{A}}$ and $\text{Game}^4_{\mathcal{A}}$

Indeed, the third equality in this derivation follows from the fact that $\mathcal{B}_1^{\mathcal{A}}(\text{seed}_{\mathbf{A}}, \mathbf{b}_u, \mathbf{a}, d_u)$ perfectly simulates $\text{Game}^3_{\mathcal{A}}$ when $u = 0$ and $\text{Game}^4_{\mathcal{A}}$ when $u = 1$.

Analysis of $\text{Game}^4_{\mathcal{A}}$ Examining $\text{Game}^4_{\mathcal{A}}$, we observe that all artifacts given to the adversary are uniformly distributed over their domain; particularly, $\hat{c}$ is uniformly distributed over R_p because the uniformity of v' is maintained under addition with (the scaled) m_u. Certainly, in this game, v' essentially constitutes a generalization of the one-time pad to the (additive) group of R_p. As such, the computed ciphertext is uniformly distributed and completely independent of all other information. This implies that an adversary against $\text{Game}^4_{\mathcal{A}}$ must randomly guess the bit u; as a result, for any adversary $\mathcal{A}$, we have $\Pr\left[\text{Game}^4_{\mathcal{A}} = 1\right] = \frac{1}{2}$.

Final Derivation Aggregating all results, we derive the security theorem formulated in Section 3.2.2. For intelligibility purposes, an intermediate result is established preceding the derivation of the security theorem.

Considering any adversary $\mathcal{A}$ and some adversaries $\mathcal{A}''$ and $\mathcal{B}_1$, the imminent derivation demonstrates a relation between, on the one hand, the difference in winning probabilities of $\mathcal{A}$ against $\text{Game}^1_{\mathcal{A}}$ and $\mathcal{A}''$ against $\text{Game}^4_{\mathcal{A}''}$, and, on the other hand, the advantage of $\mathcal{B}_1$ against an instance of the XMLWR game.

$$
\begin{aligned}
\forall_{\mathcal{A}} \exists_{\mathcal{A}',\mathcal{A}'',\mathcal{B}_1} : \left|\Pr\left[\text{Game}^1_{\mathcal{A}} = 1\right] - \Pr\left[\text{Game}^4_{\mathcal{A}''} = 1\right]\right| = & \qquad \langle \Pr\left[\text{Game}^4_{\mathcal{A}''} = 1\right] = \tfrac{1}{2} \rangle \\
\left|\Pr\left[\text{Game}^1_{\mathcal{A}} = 1\right] - \frac{1}{2}\right| = & \qquad \langle \text{Definition} \rangle \\
\mathsf{Adv}^1(\mathcal{A}) = & \qquad \langle \text{Step 2} \rangle \\
\mathsf{Adv}^2(\mathcal{A}') = & \qquad \langle \text{Step 3} \rangle \\
\mathsf{Adv}^3(\mathcal{A}'') = & \qquad \langle \text{Definition} \rangle \\
\left|\Pr\left[\text{Game}^3_{\mathcal{A}''} = 1\right] - \frac{1}{2}\right| = & \qquad \langle \Pr\left[\text{Game}^4_{\mathcal{A}''} = 1\right] = \tfrac{1}{2} \rangle
\end{aligned}
$$

$$\begin{aligned}&\left|\Pr\left[\mathrm{Game}^3_{\mathcal{A}''} = 1\right] - \Pr\left[\mathrm{Game}^4_{\mathcal{A}''} = 1\right]\right| = && \langle\text{Step 4}\rangle\\ &\mathsf{Adv}^{\text{XMLWR}}_{l,\mu,q,p}(\mathcal{B}_1)\end{aligned}$$

Utilizing this intermediate result, we derive the final result in the following manner.

$$\begin{aligned}\forall_{\mathcal{A}}\exists_{\mathcal{A}',\mathcal{B}_0,\mathcal{B}_1} : {}&\mathsf{Adv}^{\text{IND-CPA}}_{\text{Saber.PKE}}(\mathcal{A}) = && \langle\text{Initial Game}\rangle\\ &\mathsf{Adv}^0(\mathcal{A}) = && \langle\text{Definition}\rangle\\ &\left|\Pr\left[\mathrm{Game}^0_{\mathcal{A}} = 1\right] - \frac{1}{2}\right| = && \langle\Pr\left[\mathrm{Game}^4_{\mathcal{A}'} = 1\right] = \frac{1}{2}\rangle\\ &\left|\Pr\left[\mathrm{Game}^0_{\mathcal{A}} = 1\right] - \Pr\left[\mathrm{Game}^4_{\mathcal{A}'} = 1\right]\right| \leq && \langle\text{Triangle Inequality}\rangle\\ &\left|\Pr\left[\mathrm{Game}^0_{\mathcal{A}} = 1\right] - \Pr\left[\mathrm{Game}^1_{\mathcal{A}} = 1\right]\right| + \\ &\left|\Pr\left[\mathrm{Game}^1_{\mathcal{A}} = 1\right] - \Pr\left[\mathrm{Game}^4_{\mathcal{A}'} = 1\right]\right| = && \langle\text{Step 1}\rangle\\ &\mathsf{Adv}^{\text{GMLWR}}_{l,\mu,q,p}(\mathcal{B}_0) + \left|\Pr\left[\mathrm{Game}^1_{\mathcal{A}} = 1\right] - \Pr\left[\mathrm{Game}^4_{\mathcal{A}'} = 1\right]\right| = && \langle\text{Intermediate Result}\rangle\\ &\mathsf{Adv}^{\text{GMLWR}}_{l,\mu,q,p}(\mathcal{B}_0) + \mathsf{Adv}^{\text{XMLWR}}_{l,\mu,q,p}(\mathcal{B}_1)\end{aligned}$$

Naturally, compressing this derivation gives the desired result.

$$\forall_{\mathcal{A}}\exists_{\mathcal{B}_0,\mathcal{B}_1} : \mathsf{Adv}^{\text{IND-CPA}}_{\text{Saber.PKE}}(\mathcal{A}) \leq \mathsf{Adv}^{\text{GMLWR}}_{l,\mu,q,p}(\mathcal{B}_0) + \mathsf{Adv}^{\text{XMLWR}}_{l,\mu,q,p}(\mathcal{B}_1)$$

As a final remark, although no formal analysis is provided, it is evident that the running time for each of $\mathcal{B}_0$ and $\mathcal{B}_1$ is approximately equal to that of $\mathcal{A}$. In particular, excluding the calls to $\mathcal{A}$'s abstract algorithms, all employed reductions exclusively perform sequential operations that can straightforwardly be executed efficiently. As such, the proof additionally satisfies the running time requirement of the theorem.

3.2.3 Correctness

Proceeding from the discussion on security, we analyze the correctness of Saber.PKE. Intuitively, in this context, correctness refers to the probability that decrypting the encryption of a message returns this message. Here, the employed encryption and decryption procedures should respectively correspond to the considered PKE's encryption and decryption algorithms; moreover, the utilized keys should be generated with the PKE's key generation algorithm. For many public-key encryption schemes, this probability equals 1; that is, decryption of an encrypted message is guaranteed to return the original message. Nevertheless, due to the errors introduced by the modular scaling and flooring operations, this is not the case for Saber.PKE.

Concretely, in the subsequent discussion, we consider Saber.PKE's correctness with respect to two subtly different definitions: the definition utilized in Saber.PKE's original correctness proof and the definition employed by the relevant variant of the FO transformation through which Saber.KEM[11] is constructed [14, 25]. In particular, we show that these definitions are equivalent in the context of Saber.PKE; that is, a correctness analysis of Saber.PKE that utilizes one of these definitions invariably produces identical results to an analysis that uses the other definition. Furthermore, in proving the equivalence between these correctness definitions, we derive an expression that entirely determines Saber.PKE's correctness. More precisely, given a particular parameter set, it is feasible to compute the distribution of this expression exhaustively; in turn, knowledge of this distribution allows for the determination of Saber.PKE's correctness. In fact, there exists a script, designed by the authors of Saber, that does exactly this [14]. Throughout the remainder, we will refer to this script as "Saber's script".

As with any PKE scheme, determining the correctness of Saber.PKE is imperative to validate that the scheme's algorithms sufficiently conform to the desired relation with respect to the possible inputs; indeed, as alluded to above, this relation intuitively denotes whether encrypting and,

[11]As its name suggests, Saber.KEM refers to Saber's KEM.

subsequently, decrypting any valid message through the scheme's algorithms returns this message with sufficient probability. However, in the case of Saber.PKE, there exists an additional rationale for determining its correctness. Namely, in consequence of (the variant of) the FO transformation utilized to construct Saber.KEM from Saber.PKE, the security and correctness properties of Saber.KEM are entirely contingent on the correctness property of Saber.PKE, i.e., under the definition employed by this FO transformation [14, 25]. For these reasons, it is crucial to analyze Saber.PKE's correctness with respect to both relevant definitions.

The discussion on Saber.PKE's correctness ensues as follows. First, we provide an alternative specification of Saber.PKE and prove it equivalent to the original specification given in Section 3.2.1. Then, with regards to the relevant context, we elaborate on the two above-mentioned correctness definitions. Subsequently, utilizing the alternative specification of Saber.PKE, we show the equivalence between these correctness definitions and, moreover, derive the expression with which Saber's script exhaustively computes Saber.PKE's correctness. Lastly, we mention the implications of the performed correctness analysis and present some concrete correctness values of Saber.PKE and Saber.KEM for several customary parameter sets.

Before advancing, it is helpful to recollect Saber's parameter-related definitions and requirements specified in Section 3.1. In particular, recall that $t = 2^{\epsilon_t}$, $p = 2^{\epsilon_p}$, and $q = 2^{\epsilon_q}$, where $0 < \epsilon_t + 1 < \epsilon_p < \epsilon_q$; consequently, $0 < 2 \cdot t < p < q$ and $2 \cdot t \mid p \mid q$.

Alternative Specification of Saber.PKE

Prior to the actual correctness analysis, we define an alternative specification of Saber.PKE and prove it equivalent to the original specification provided in Section 3.2.1. The motive for constructing this alternative specification is the simplification of both the manual analysis and the corresponding formal verification effort. Specifically, while the original specification performs computations in several different polynomial quotient rings (e.g., R_q and R_p), the alternative specification exclusively operates in R_q[12]. Concretely, we accomplish this by lifting the relevant computations to R_q through the (modular) scaling of the appropriate values. This alteration makes the manual correctness analysis more comprehensible by reducing the need to be mindful of the interpretation of the considered elements, allowing for a less involved definition and analysis of the induced errors and their consequences; furthermore, this change simplifies the corresponding formal verification effort by decreasing the number of necessary types and type conversions in the operations. Despite these benefits concerning the correctness analysis, the alternative specification is not employed as the standard specification of Saber.PKE; the principal reason for this is that, relative to the customary (i.e., the original) specification, the alternative specification is slightly more intricate. Throughout, the alternative specification of the scheme and its algorithms are referred to as Saber.PKEA = ($\mathsf{Saber.KeyGenA}$, $\mathsf{Saber.EncA}$, $\mathsf{Saber.DecA}$).

In the current context, two schemes are equivalent if each algorithm in one of the schemes is equivalent to exactly one algorithm of the other scheme. In turn, two algorithms are equivalent if their input and output domains are equal and, given the same input, they return identical output. Here, in case the collated algorithms are probabilistic, "identical output" refers to "an equal probability of producing the same output". As such, if the analysis of a certain property merely regards the input and output behavior of a scheme's algorithms, substituting the considered algorithms with equivalent ones does not influence the analysis's result. Indeed, the subsequent correctness analysis of Saber.PKE belongs to this category; that is, this analysis only considers the input and output behavior of Saber.PKE's algorithms. Hence, if Saber.PKE and Saber.PKEA are equivalent, Saber.PKEA's algorithms may replace Saber.PKE's algorithms in the correctness analysis of Saber.PKE without invalidating the result.

[12]Actually, several morphisms between different algebraic structures form exceptions to this exclusive operation in R_q; primarily, this concerns the modular scaling and flooring operators.

Regarding the actual specification of Saber.PKEA, Algorithm 4 provides the specification of Saber.KeyGenA. As we can see, this algorithm is identical to its original counterpart specified in Algorithm 1. Certainly, this is because Saber.KeyGen already performs all relevant computations in R_q. Nevertheless, for reasons of completeness, Saber.KeyGenA is presented separately here.

Algorithm 4 Alternative Specification of Saber's Key Generation Algorithm

```
1: procedure Saber.KeyGenA()
2:     $\text{seed}_{\mathbf{A}} \leftarrow\!\$\ \mathcal{U}(\{0,1\}^{256})$
3:     $\mathbf{A} \leftarrow \mathsf{gen}(\text{seed}_{\mathbf{A}})$
4:     $\mathbf{s} \leftarrow\!\$\ \beta_\mu(R_q^{l\times 1})$
5:     $\mathbf{b} \leftarrow \lfloor \mathbf{A}\cdot\mathbf{s} + \mathbf{h} \rceil_{q\to p}$
6:     return $\mathsf{pk} := (\text{seed}_{\mathbf{A}}, \mathbf{b}),\ \mathsf{sk} := \mathbf{s}$
7: end procedure
```

Next, Algorithm 5 provides the specification of Saber.EncA. Comparing this specification to the specification of Saber.Enc given in Algorithm 2, we see that both algorithms have an identical input and output domain. Furthermore, the only differences between these algorithms are the following.

- Before the computation of v', Saber.EncA performs a modular scaling of $\mathbf{b}$ from modulo p to modulo q, i.e., $\lfloor \mathbf{b} \rceil_{p\to q}$. Saber.Enc does not perform this operation.

- In the computation of v', Saber.EncA does not reduce $\mathbf{s}'$ and h_1 modulo p, while Saber.Enc does.

- In the computation of v', Saber.EncA multiplies h_1 with the constant polynomial $\frac{q}{p} \in R_q$, while Saber.Enc does not.

- In the computation of c_m, Saber.EncA utilizes q as the target and source modulus for, respectively, the modular scaling of m and the modular scaling and flooring of $v' + \lfloor m \rceil_{2\to q}$. Contrarily, in these instances, Saber.Enc uses p instead of q.

As a consequence of the first, second and last difference, Saber.EncA performs the computation of v' and c_m (before the modular scaling and flooring) in R_q. Additionally, the third difference ensures that the eventual output of Saber.EncA is identical to that of Saber.Enc; in particular, it guarantees that Saber.EncA's c_m is identical to the c_m of Saber.Enc. Namely, since none of the above-mentioned differences consider the other part of the output, i.e., $\mathbf{b}'$, the manner in which both specifications compute this vector is exactly the same; as such, this part of the output is trivially equal between Saber.Enc and Saber.EncA. For this reason, the imminent discussion and derivations merely show the equality between the algorithms' initial part of the output, i.e., c_m.

As a convenient intermediate step to proving the equality of the c_m from Saber.EncA and Saber.Enc, we show that the ϵ_p most significant bits of the computed v' are identical between these specifications. To see why this is true, consider Saber.EncA's computation of v' without the multiplication of h_1 with $\frac{q}{p}$; moreover, replace $\mathbf{b}_q$ with $\mathbf{b}$, interpreted as an element of R_q. In other words, consider the computation of $\mathbf{b}^T \cdot \mathbf{s}' + h_1$ in R_q. Then, following the same reasoning as in the third step of the security proof presented in Section 3.2.2, performing this computation in R_p produces a result of which each coefficient's ϵ_p (total) bits are identical to the ϵ_p *least* significant bits of its counterpart in R_q. Consequently, multiplying the result in R_q with $\frac{q}{p} \in R_q$ yields a value of which each coefficient's ϵ_p *most* significant bits precisely equal the corresponding coefficient's ϵ_p (total) bits in R_p. More precisely, let $a_{\epsilon_q-1} \dots a_{\epsilon_p-1} \dots a_0$ denote the binary representation of a coefficient

Algorithm 5 Alternative Specification of Saber's Encryption Algorithm

```
1: procedure Saber.EncA(pk := (seed_A, b), m)
2:     $\mathbf{A} \leftarrow \mathsf{gen}(\mathrm{seed}_{\mathbf{A}})$
3:     $\mathbf{s}' \leftarrow\$ \; \beta_\mu(R_q^{l\times 1})$
4:     $\mathbf{b}' \leftarrow \lfloor \mathbf{A}^T \cdot \mathbf{s}' + \mathbf{h} \rfloor_{q \to p}$
5:     $\mathbf{b}_q \leftarrow \lfloor \mathbf{b} \rfloor_{p \to q}$
6:     $v' \leftarrow \mathbf{b}_q^T \cdot \mathbf{s}' + \frac{q}{p} \cdot h_1$
7:     $c_m \leftarrow \lfloor v' + \lfloor m \rfloor_{2 \to q} \rfloor_{q \to 2 \cdot t}$
8:     return $c := (c_m, \mathbf{b}')$
9: end procedure
```

from (the result of) $\mathbf{b}^T \cdot \mathbf{s}' + h_1$ in R_q; accordingly, $a_{\epsilon_p - 1} \dots a_0$ denotes the binary representation of the corresponding coefficient in R_p. Then, the multiplication of a coefficient from the result in R_q by $\frac{q}{p} \in \mathbb{Z}_q$ is computed as follows.

$$\begin{aligned}\frac{q}{p} \cdot a_{\epsilon_q - 1} \dots a_{\epsilon_p - 1} \dots a_0 \bmod q &= 2^{\epsilon_q - \epsilon_p} \cdot a_{\epsilon_q - 1} \dots a_{\epsilon_p - 1} \dots a_0 \bmod 2^{\epsilon_q} \\ &= a_{\epsilon_q - 1} \dots a_{\epsilon_p - 1} \dots a_0 0^{\epsilon_q - \epsilon_p} \bmod 2^{\epsilon_q} \\ &= a_{\epsilon_p - 1} \dots a_0 0^{\epsilon_q - \epsilon_p}\end{aligned}$$

As elaborated on before in the context of similar computations, the second equality is a consequence of the fact that multiplication by an integral power-of-two induces a left bit-shift by a number of bits that is equal to the exponent's value, i.e., $\epsilon_q - \epsilon_p$ in this case. Moreover, reduction modulo $q = 2^{\epsilon_q}$ effectively discards all bits *more* significant than the ϵ_q-th bit, giving rise to the last equality.

Reconsidering the above-discussed multiplication of $\mathbf{b}^T \cdot \mathbf{s}' + h_1$ with $\frac{q}{p}$ in R_q, i.e., $\frac{q}{p} \cdot (\mathbf{b}^T \cdot \mathbf{s}' + h_1)$, we rewrite this as follows.

$$\begin{aligned}\frac{q}{p} \cdot (\mathbf{b}^T \cdot \mathbf{s}' + h_1) &= \frac{q}{p} \cdot (\mathbf{b}^T \cdot \mathbf{s}') + \frac{q}{p} \cdot h_1 \\ &= (\frac{q}{p} \cdot \mathbf{b}^T) \cdot \mathbf{s}' + \frac{q}{p} \cdot h_1 \\ &= (\frac{q}{p} \cdot \mathbf{b})^T \cdot \mathbf{s}' + \frac{q}{p} \cdot h_1\end{aligned}$$

Recall that, since $\mathbf{b}$ is the result of a modular scaling and flooring operation with target modulus p, this vector is entirely comprised of polynomials with coefficients in the discrete range $[0, 2^{\epsilon_p} - 1]$. Consequently, $\frac{q}{p} \cdot \mathbf{b} = 2^{\epsilon_q - \epsilon_p} \cdot \mathbf{b}$ consists of polynomials with coefficients in the set $\{0 \cdot 2^{\epsilon_q - \epsilon_p}, 1 \cdot 2^{\epsilon_q - \epsilon_p}, 2 \cdot 2^{\epsilon_q - \epsilon_p}, \dots, (2^{\epsilon_p} - 1) \cdot 2^{\epsilon_q - \epsilon_p}\}$. As such, because each coefficient is an integer, the flooring function reduces to the identity function; moreover, since the set in which each coefficient resides is a subset of the discrete range $[0, 2^{\epsilon_q - 1}]$, reduction modulo $q = 2^{\epsilon_q}$ similarly reduces to the identity function. Utilizing these observations, we rewrite the above $(\frac{q}{p} \cdot \mathbf{b})^T \cdot \mathbf{s}' + \frac{q}{p} \cdot h_1$ even further, as shown below.

$$\begin{aligned}(\frac{q}{p} \cdot \mathbf{b})^T \cdot \mathbf{s}' + \frac{q}{p} \cdot h_1 &= (\left\lfloor \frac{q}{p} \cdot \mathbf{b} \right\rfloor \bmod q)^T \cdot \mathbf{s}' + \frac{q}{p} \cdot h_1 \\ &= (\lfloor \mathbf{b} \rfloor_{p \to q})^T \cdot \mathbf{s}' + \frac{q}{p} \cdot h_1 \\ &= \mathbf{b}_q^T \cdot \mathbf{s}' + \frac{q}{p} \cdot h_1\end{aligned}$$

Patently, the final result, i.e., $\mathbf{b}_q^T \cdot \mathbf{s}' + \frac{q}{p} \cdot h_1$, precisely corresponds to the computation of v' in Saber.EncA. Thus, by transitivity, we deduce that $\frac{q}{p} \cdot (\mathbf{b}^T \cdot \mathbf{s}' + h_1) = \mathbf{b}_q^T \cdot \mathbf{s}' + \frac{q}{p} \cdot h_1 = v'$, where v'

refers to the v' from Saber.EncA. Furthermore, incorporating the preceding result, we see that the ϵ_p most significant bits of a coefficient from $\frac{q}{p} \cdot (\mathbf{b}^T \cdot \mathbf{s}' + h_1)$ in R_q are identical to the ϵ_p (total) bits of the corresponding coefficient from $\mathbf{b}^T \cdot \mathbf{s}' + h_1$ in R_p. Following, since this latter computation corresponds to the v' in Saber.Enc, we conclude that the ϵ_p most significant bits of each coefficient of the v' in Saber.EncA are equal to the ϵ_p (total) bits of the corresponding coefficient of the v' in Saber.Enc.

Employing the results obtained thus far, we derive the value of c_m's coefficients for both specifications. Specifically, using b_0 to denote the binary representation of a coefficient from m, this derivation is as follows for Saber.EncA.

$$\begin{aligned}
\lfloor a_{\epsilon_p-1} \ldots a_0 0^{\epsilon_q-\epsilon_p} + \lfloor b_0 \rceil_{2\to q} \rceil_{q\to 2\cdot t} &= \lfloor a_{\epsilon_p-1} \ldots a_0 0^{\epsilon_q-\epsilon_p} + b_0 0^{\epsilon_q-1} \rceil_{q\to 2\cdot t} \\
&= \lfloor (a_{\epsilon_p-1} + b_0) \ldots a_0 0^{\epsilon_q-\epsilon_p} \rceil_{q\to 2\cdot t} \\
&= (a_{\epsilon_p-1} + b_0) \ldots a_{\epsilon_p-\epsilon_t-1}
\end{aligned}$$

As in Section 3.2.2, $(a_{\epsilon_p-1} + b_0)$ represents the bit value resulting from the addition of bits a_{ϵ_p-1} and b_0 (modulo 2); moreover, because the complete addition of $a_{\epsilon_p-1} \ldots a_0 0^{\epsilon_q-\epsilon_p}$ and $b_0 0^{\epsilon_q-1}$ is carried out in $\mathbb{Z}_q$, the implicit reduction modulo q effectively removes the $(\epsilon_q + 1)$-th bit that may be produced through a potential carry. The final equality holds because $(a_{\epsilon_p-1} + b_0) \ldots a_0 0^{\epsilon_q-\epsilon_p}$ consists of ϵ_q bits and the $\lfloor \cdot \rceil_{q\to 2\cdot t}$ operator induces a right bit-shift of $\epsilon_q - (\epsilon_t + 1)$ bits; hence, after the application of this operator, only the $\epsilon_q - (\epsilon_q - (\epsilon_t + 1)) = \epsilon_t + 1$ most significant bits remain.

Utilizing the same notation as above, the analogous derivation for Saber.Enc is given below.

$$\begin{aligned}
\lfloor a_{\epsilon_p-1} \ldots a_0 + \lfloor b_0 \rceil_{2\to p} \rceil_{p\to 2\cdot t} &= \lfloor a_{\epsilon_p-1} \ldots a_0 + b_0 0^{\epsilon_p-1} \rceil_{p\to 2\cdot t} \\
&= \lfloor (a_{\epsilon_p-1} + b_0) \ldots a_0 \rceil_{p\to 2\cdot t} \\
&= (a_{\epsilon_p-1} + b_0) \ldots a_{\epsilon_p-\epsilon_t-1}
\end{aligned}$$

Evidently, this derivation is nearly identical to the previous one, only differing in the number of trailing zero bits in the elements' binary representation and in one of the moduli for the modular scaling and flooring operators. Nevertheless, these differences are precisely such that the resulting values of both derivations are exactly equal. This implies that each coefficient of the c_m from Saber.Enc is equal to its counterpart of the c_m from Saber.EncA; in turn, it follows that the c_m from both specifications are equal. Combining this with the formerly acquired results, we conclude that Saber.Enc and Saber.EncA are equivalent.

Finally, Algorithm 6 contains the specification of Saber.DecA. As is trivially confirmed, the input and output domains of both Saber.Dec and Saber.DecA are identical. Furthermore, the differences between these specifications are quite similar to the differences between Saber.Enc and Saber.EncA; specifically, the differences between Saber.Dec and Saber.DecA are the following.

- Before the computation of v, Saber.DecA performs a modular scaling of b' from modulo p to modulo q, i.e., $\lfloor \mathsf{b}' \rceil_{p\to q}$. Saber.Dec does not perform this operation.

- Before the computation of m', Saber.DecA performs a modular scaling of c_m from modulo $2 \cdot t$ to modulo q, i.e., $\lfloor c_m \rceil_{2\cdot t\to q}$. Saber.Dec does not perform this operation.

- In the computations of v and m', Saber.DecA does not reduce s', h_1, and h_2 modulo p, while Saber.Dec does.

- In the computations of v and m', respectively, Saber.DecA multiplies h_1 and h_2 with the constant polynomial $\frac{q}{p} \in R_q$, while Saber.Dec does not.

- In the computation of m', $\mathsf{Saber.DecA}$ utilizes q as the source modulus for the modular scaling and flooring of $v - \lfloor c_m \rfloor_{2\cdot t \to q} + \frac{q}{p} \cdot h_2$. Contrarily, $\mathsf{Saber.Dec}$ uses p as the source modulus for its analogous operation.

As a consequence of the resemblance between these differences and the ones for $\mathsf{Saber.Enc}$ and $\mathsf{Saber.EncA}$, the output analysis of $\mathsf{Saber.Dec}$ and $\mathsf{Saber.DecA}$ is comparable to that of the encryption algorithms. Particularly, due to the similarities between the computations of c_m and m', showing that m' is identical between $\mathsf{Saber.Dec}$ and $\mathsf{Saber.DecA}$ is tantamount to showing c_m is identical between $\mathsf{Saber.Enc}$ and $\mathsf{Saber.EncA}$. As such, in the subsequent discussion on the equality between the outputs of $\mathsf{Saber.Dec}$ and $\mathsf{Saber.DecA}$, we will frequently refer back to the analogous discussion on the outputs of $\mathsf{Saber.Enc}$ and $\mathsf{Saber.EncA}$.

Algorithm 6 Alternative Specification of Saber's Decryption Algorithm

```
1: procedure $\mathsf{Saber.DecA}(\mathsf{sk} := \mathbf{s},\ c := (c_m, \mathbf{b}'))$
2:     $\mathbf{b}'_q \leftarrow \lfloor \mathbf{b}' \rfloor_{p \to q}$
3:     $v \leftarrow \mathbf{b}'^T_q \cdot \mathbf{s} + \frac{q}{p} \cdot h_1$
4:     $c_{mq} \leftarrow \lfloor c_m \rfloor_{2\cdot t \to q}$
5:     $m' \leftarrow \lfloor v - c_{mq} + \frac{q}{p} \cdot h_2 \rfloor_{q \to 2}$
6:     return $m'$
7: end procedure
```

First, we rearrange the computation of m' in $\mathsf{Saber.Dec}$ as follows.

$$\lfloor v - \lfloor c_m \rfloor_{2\cdot t \to p} + (h_2 \bmod p) \rfloor_{p \to 2} = \lfloor (v + (h_2 \bmod p)) - \lfloor c_m \rfloor_{2\cdot t \to p} \rfloor_{p \to 2}$$

Similarly, we rewrite the computation of m' in $\mathsf{Saber.DecA}$ as illustrated below.

$$\begin{aligned}\lfloor v - c_{mq} + \frac{q}{p} \cdot h_2 \rfloor_{q \to 2} &= \lfloor (v + \frac{q}{p} \cdot h_2) - c_{mq} \rfloor_{q \to 2} \\ &= \lfloor (v + \frac{q}{p} \cdot h_2) - \lfloor c_m \rfloor_{2\cdot t \to q} \rfloor_{q \to 2}\end{aligned}$$

Then, letting $a_{\epsilon_p - 1} \dots a_0$ denote the binary representation of a coefficient of $(v + (h_2 \bmod p))$ from $\mathsf{Saber.Dec}$, we extend the reasoning applied to v' in the output analysis of $\mathsf{Saber.Enc}$ and $\mathsf{Saber.EncA}$ to also consider the addition of h_2. From this reasoning extension, we derive that the binary representation of a coefficient of $v + \frac{q}{p} \cdot h_2$ from $\mathsf{Saber.DecA}$ equals $a_{\epsilon_p - 1} \dots a_0 0^{\epsilon_q - \epsilon_p}$.

Utilizing the above-derived binary representation, the ensuing derivation shows how $\mathsf{Saber.DecA}$ computes a coefficient of m'; in this derivation, the binary representation of a coefficient of c_m is denoted by $d_{\epsilon_t} \dots d_0$.

$$\begin{aligned}\lfloor a_{\epsilon_p - 1} \dots a_0 0^{\epsilon_q - \epsilon_p} - \lfloor d_{\epsilon_t} \dots d_0 \rfloor_{2\cdot t \to q} \rfloor_{q \to 2} &= \lfloor a_{\epsilon_p - 1} \dots a_0 0^{\epsilon_q - \epsilon_p} - d_{\epsilon_t} \dots d_0 0^{\epsilon_q - \epsilon_t - 1} \rfloor_{q \to 2} \\ &= \lfloor (10^{\epsilon_q} + a_{\epsilon_p - 1} \dots a_0 0^{\epsilon_q - \epsilon_p}) - d_{\epsilon_t} \dots d_0 0^{\epsilon_q - \epsilon_t - 1} \rfloor_{q \to 2} \\ &= \lfloor 1 a_{\epsilon_p - 1} \dots a_0 0^{\epsilon_q - \epsilon_p} - d_{\epsilon_t} \dots d_0 0^{\epsilon_q - \epsilon_t - 1} \rfloor_{q \to 2} \\ &= \lfloor f_{\epsilon_p} f_{\epsilon_p - 1} \dots f_{\epsilon_p - \epsilon_t} (a_{\epsilon_p - \epsilon_t - 1} - d_0) \dots a_0 0^{\epsilon_q - \epsilon_p} \rfloor_{q \to 2} \\ &= \lfloor f_{\epsilon_p - 1} \dots f_{\epsilon_p - \epsilon_t} (a_{\epsilon_p - \epsilon_t - 1} - d_0) \dots a_0 0^{\epsilon_q - \epsilon_p} \rfloor_{q \to 2} \\ &= f_{\epsilon_p - 1}\end{aligned}$$

Here, the second equality follows from the fact that any element $x \in \mathbb{Z}_q$ is congruent to $q + x$; in particular, $a_{\epsilon_p - 1} \dots a_0 0^{\epsilon_q - \epsilon_p}$ is congruent to $q + a_{\epsilon_p - 1} \dots a_0 0^{\epsilon_q - \epsilon_p}$. Consequently, $a_{\epsilon_p - 1} \dots a_0 0^{\epsilon_q - \epsilon_p}$ may be substituted by $q + a_{\epsilon_p - 1} \dots a_0 0^{\epsilon_q - \epsilon_p}$ without altering the outcome[13]. The rationale for this

[13]Since $q = 2^{\epsilon_q}$, the binary representation of q is 10^{ϵ_q}; to remain consistent with the representation of the other elements, the derivation utilizes this binary representation.

replacement is the prevention of a potential underflow, i.e., modular wrap-around, in the case that $a_{\epsilon_p-1}\dots a_0 0^{\epsilon_q-\epsilon_p} < d_{\epsilon_t}\dots d_0 0^{\epsilon_q-\epsilon_t-1}$. Certainly, explicitly accounting for such underflow would require a significantly more intricate derivation and explanation. Furthermore, in the above derivation, the f_i represent bits that result from the subtraction $1a_{\epsilon_p-1}\dots a_0 0^{\epsilon_q-\epsilon_p} - d_{\epsilon_t}\dots d_0 0^{\epsilon_q-\epsilon_t-1}$ and are potentially affected by borrows. For reasons of clarity, the index of each f_i matches the index of the corresponding a_i that it replaced after the subtraction; e.g., after the subtraction, f_{ϵ_p-1} has the same position as a_{ϵ_p-1} had before the subtraction. The only exception to this is f_{ϵ_p}, the (ϵ_q+1)-th bit of the subtraction result; indeed, this bit signifies the result of a potential borrow from the 1 at the (ϵ_q+1)-th position in $1a_{\epsilon_p-1}\dots a_0 0^{\epsilon_q-\epsilon_p}$. In contrast to the f_i bits, $(a_{\epsilon_p-\epsilon_t-1}-d_0)$ denotes a (single) bit value arising from the same subtraction, but which can not be affected by borrows; in fact, considering the position of this bit in the subtraction result, we see that its value must be equal to the subtraction of d_0 from $a_{\epsilon_p-\epsilon_t-1}$ (modulo 2). Lastly, due to the earlier substitution of $a_{\epsilon_p-1}\dots a_0 0^{\epsilon_q-\epsilon_p}$ with the congruent $q + a_{\epsilon_p-1}\dots a_0 0^{\epsilon_q-\epsilon_p}$, which is not fully reduced, the subtraction result is not fully reduced either. To preclude an unnecessary elaboration regarding the application of modular scaling and flooring operators on elements that are not fully reduced, we replace the subtraction result by a congruent value that is (fully) reduced modulo q, i.e., a canonical representative, before applying the $\lfloor\cdot\rfloor_{q\to 2}$ operator. Since reduction modulo q effectively discards all bits *more* significant than the ϵ_q-th bit, this congruent value is identical to the original subtraction result without the (ϵ_q+1)-th bit, i.e. the f_{ϵ_p} bit.

Retaining the same notation for the binary representation of a coefficient of c_m, we particularize the computation of a coefficient of m' in Saber.Dec. For this derivation, remark that $a_{\epsilon_p-1}\dots a_0$ is congruent to $p + a_{\epsilon_p-1}\dots a_0 = 10^{\epsilon_p} + a_{\epsilon_p-1}\dots a_0$; certainly, this is because $a_{\epsilon_p-1}\dots a_0 \in \mathbb{Z}_p$

$$\begin{aligned}
\lfloor a_{\epsilon_p-1}\dots a_0 - \lfloor d_{\epsilon_t}\dots d_0\rfloor_{2\cdot t\to p}\rfloor_{p\to 2} &= \lfloor a_{\epsilon_p-1}\dots a_0 - d_{\epsilon_t}\dots d_0 0^{\epsilon_p-\epsilon_t-1}\rfloor_{p\to 2}\\
&= \lfloor (10^{\epsilon_p} + a_{\epsilon_p-1}\dots a_0) - d_{\epsilon_t}\dots d_0 0^{\epsilon_p-\epsilon_t-1}\rfloor_{p\to 2}\\
&= \lfloor 1a_{\epsilon_p-1}\dots a_0 - d_{\epsilon_t}\dots d_0 0^{\epsilon_p-\epsilon_t-1}\rfloor_{p\to 2}\\
&= \lfloor f_{\epsilon_p} f_{\epsilon_p-1}\dots f_{\epsilon_p-\epsilon_t}(a_{\epsilon_p-\epsilon_t-1}-d_0)\dots a_0\rfloor_{p\to 2}\\
&= \lfloor f_{\epsilon_p-1}\dots f_{\epsilon_p-\epsilon_t}(a_{\epsilon_p-\epsilon_t-1}-d_0)\dots a_0\rfloor_{p\to 2}\\
&= f_{\epsilon_p-1}
\end{aligned}$$

As we can see, Saber.Dec's computation of (each coefficient of) m' differs in a few aspects from the analogous computation of Saber.DecA. Despite these differences however, both Saber.Dec and Saber.DecA subtract the values constituted by the variable bits in an identical manner; that is, they both essentially subtract $d_{\epsilon_t}\dots d_0$ from $1a_{\epsilon_p-1}\dots a_0$, albeit with a deviating number of trailing zero bits. Consequently, the results of these subtractions are also precisely equal in their variable bits, i.e., $f_{\epsilon_p} f_{\epsilon_p-1}\dots f_{\epsilon_p-\epsilon_t}(a_{\epsilon_p-\epsilon_t-1}-d_0)\dots a_0$, and solely vary in the number of trailing zero bits. Eventually, the difference in trailing zero bits is exactly compensated for by the different modular scaling and flooring operators; as a result, the final values produced for each coefficient of m', and hence m' itself, are identical between Saber.Dec and Saber.DecA. Ergo, given the same input, both specifications produce identical output. At last, we conclude that Saber.Dec and Saber.DecA are equivalent.

In conclusion, since the key generation, encryption, and decryption algorithms of Saber.PKE and Saber.PKEA are all pair-wise equivalent, the schemes themselves are also equivalent.

Correctness Definitions

As aforementioned, we analyze the correctness of Saber.PKE with respect to two slightly different definitions: the definition employed by Saber.PKE's original correctness proof and the definition utilized in the relevant variant of the FO transformation [14,25]. Throughout the remainder, these definitions are referred to as "standard correctness" and "FO-correctness", respectively.

First, standard correctness intuitively refers to the minimal probability that encrypting and, subsequently, decrypting a message from the message space gives back this original message. Naturally, in the context of Saber.PKE, the encrypting and decrypting procedures must accordingly adhere to the specifications of $\mathsf{Saber.Enc}$ and $\mathsf{Saber.Dec}$; additionally, the public and secret key utilized by these procedures must be produced in accordance with $\mathsf{Saber.KeyGen}$. More formally, regarding Saber.PKE, standard correctness refers to the $1 - \delta \in [0, 1]$ such that for any $m \in \mathcal{M}$, the following probability statement holds.

$$\Pr[\mathsf{Saber.Dec}(\mathsf{sk}, \mathsf{Saber.Enc}(\mathsf{pk}, m)) = m \mid (\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Saber.KeyGen}()] \geq 1 - \delta$$

In agreement with the preceding intuitive description of standard correctness, this statement essentially encompasses that for any message $m \in \mathcal{M}$, sequentially executing $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Saber.KeyGenA}()$, $c \leftarrow \mathsf{Saber.EncA}(\mathsf{pk}, m)$, and $m' \leftarrow \mathsf{Saber.DecA}(\mathsf{sk}, c)$ results in $m' = m$ with a probability of at least $1 - \delta$. As such, this correctness definition allows for a convenient formalization in terms of a probabilistic program; Figure 3.13 provides this formalization on its left side. In terms of this

```
$\text{PProg}^{\text{STDCOR}}_{\text{Saber.PKE}}(m)$
1 :  $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Saber.KeyGen}()$
2 :  $c \leftarrow \mathsf{Saber.Enc}(\mathsf{pk}, m)$
3 :  $m' \leftarrow \mathsf{Saber.Dec}(\mathsf{sk}, c)$
4 :  $\textbf{return } (m' = m)$
```

```
$\text{PProg}^{\text{STDCOR}}_{\text{Saber.PKEA}}(m)$
1 :  $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Saber.KeyGenA}()$
2 :  $c \leftarrow \mathsf{Saber.EncA}(\mathsf{pk}, m)$
3 :  $m' \leftarrow \mathsf{Saber.DecA}(\mathsf{sk}, c)$
4 :  $\textbf{return } (m' = m)$
```

Figure 3.13: Formalization of Standard Correctness for Saber.PKE (Left) and Saber.PKEA (Right)

probabilistic program, we can rewrite the above probability statement to the following equivalent statement.

$$\Pr\left[\text{PProg}^{\text{STDCOR}}_{\text{Saber.PKE}}(m) = 1\right] \geq 1 - \delta$$

As elaborated on before, the algorithms of Saber.PKE may, in the correctness analysis, be used interchangeably with their counterparts from Saber.PKEA; particularly, these algorithms are interchangeable in $\text{PProg}^{\text{STDCOR}}_{\text{Saber.PKE}}(m)$. Replacing each algorithm of Saber.PKE in this program by its equivalent counterpart from Saber.PKEA gives rise to the probabilistic program presented on the right side of Figure 3.13; accordingly, the corresponding probability statement changes to the statement below.

$$\Pr\left[\text{PProg}^{\text{STDCOR}}_{\text{Saber.PKEA}}(m) = 1\right] \geq 1 - \delta$$

Finally, FO-correctness is quite similar to standard correctness; in particular, FO-correctness merely differs from standard correctness with respect to the selection of the considered message. Akin to standard correctness, FO-correctness can be formalized through either a single probability statement or a probabilistic program combined with a probability statement. Nevertheless, since the subsequent correctness analysis only uses the latter formalization, we solely discuss the formalization in terms of the probabilistic program. In contrast to its counterpart from standard correctness, this formalization employs an adversary and, as such, is considered a game. Moreover, this adversary is unrestricted with regards to its resources and, given a public key and secret key as input, produces a message. Considering any such adversary $\mathcal{A}$, Figure 3.14 defines the FO-correctness game for both Saber.PKE and Saber.PKEA; indeed, due to the equivalence between Saber.PKE and Saber.PKEA, these games are also equivalent and may be used interchangeably. Lastly, analogously to Saber.PKE's standard correctness, the FO-correctness of Saber.PKE refers to the $1 - \delta \in [0, 1]$ such that for any valid adversary $\mathcal{A}$, the following probability statement holds.

$$\Pr\left[\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{Saber.PKE}} = 1\right] \geq 1 - \delta$$

Naturally, replacing $\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{Saber.PKE}}$ with $\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{Saber.PKEA}}$, we obtain the equivalent probability statement corresponding to the FO-correctness of Saber.PKEA.

```
Game^{FOCOR}_{A,Saber.PKE}
1 : $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Saber.KeyGen}()$
2 : $m \leftarrow \mathcal{A}(\mathsf{pk}, \mathsf{sk})$
3 : $c \leftarrow \mathsf{Saber.Enc}(\mathsf{pk}, m)$
4 : $m' \leftarrow \mathsf{Saber.Dec}(\mathsf{sk}, c)$
5 : return $(m' = m)$
```

```
Game^{FOCOR}_{A,Saber.PKEA}
1 : $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Saber.KeyGenA}()$
2 : $m \leftarrow \mathcal{A}(\mathsf{pk}, \mathsf{sk})$
3 : $c \leftarrow \mathsf{Saber.EncA}(\mathsf{pk}, m)$
4 : $m' \leftarrow \mathsf{Saber.DecA}(\mathsf{sk}, c)$
5 : return $(m' = m)$
```

Figure 3.14: Formalization of FO-Correctness for Saber.PKE (Left) and Saber.PKEA (Right)

Correctness Equivalence and Error Expression

Having introduced the relevant correctness definitions, we proceed by proving their equivalence with respect to Saber.PKE; moreover, in this endeavor, we derive the aforementioned expression that allows for the exhaustive computation of Saber.PKE's concrete correctness value for a certain parameter set.

Foremost, concerning $\text{PProg}^{\text{STDCOR}}_{\text{Saber.PKEA}}$ and $\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{Saber.PKEA}}$, we reiterate that both programs essentially verify whether, after the sequential execution of $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Saber.KeyGenA}()$, $c \leftarrow \mathsf{Saber.EncA}(\mathsf{pk}, m)$, and $m' \leftarrow \mathsf{Saber.DecA}(\mathsf{sk}, c)$, m' is equal to m. As such, given some $m \in \mathcal{M}$, we can derive the expression that determines whether $m' = m$ by considering the specifications of Saber.PKEA's algorithms. Nevertheless, before the derivation of this expression, we define several error terms. These error terms capture the errors introduced by the modular scaling and flooring operations in Saber.PKEA's algorithms. Ultimately, as the remainder will show, the expression derived for the verification of $m = m'$ exclusively depends on these error terms; henceforth, we refer to this expression as "error expression".

The first error term relates to $\mathbf{A} \cdot \mathbf{s}$ and $\mathbf{b}_q$; in particular, this error term, $\mathbf{err}_{\mathbf{b}_q}$, represents the error of $\mathbf{b}_q$ relative to $\mathbf{A} \cdot \mathbf{s}$, as defined below.

$$\mathbf{err}_{\mathbf{b}_q} = \mathbf{b}_q - \mathbf{A} \cdot \mathbf{s} = \lfloor \lfloor \mathbf{A} \cdot \mathbf{s} + \mathbf{h} \rfloor_{q \to p} \rfloor_{p \to q} - \mathbf{A} \cdot \mathbf{s}$$

Here, since both $\mathbf{b}_q$ and $\mathbf{A} \cdot \mathbf{s}$ are elements of $R_q^{l \times 1}$, so is $\mathbf{err}_{\mathbf{b}_q}$. Furthermore, the coefficients of (the entries of) this error term all lie in the discrete range $(-\frac{q}{2 \cdot p}, \frac{q}{2 \cdot p}]$, centered around zero[14]; this can be deduced as follows. First, because $\lfloor \mathbf{A} \cdot \mathbf{s} + \mathbf{h} \rfloor_{q \to p}$ is equivalent to $\lfloor \mathbf{A} \cdot \mathbf{s} \rceil_{q \to p}$, performing $\lfloor \mathbf{A} \cdot \mathbf{s} + \mathbf{h} \rfloor_{q \to p}$ potentially adds $2^{\epsilon_q - \epsilon_p}$ to the coefficients of (the entries of) $\mathbf{A} \cdot \mathbf{s}$ depending on a coefficient's $\epsilon_q - \epsilon_p$ least significant bits. Specifically, if these $\epsilon_q - \epsilon_p$ least significant bits correspond to a value that is closer to 0 than to $2^{\epsilon_q - \epsilon_p}$, nothing is added to the coefficient; otherwise, i.e., if these bits correspond to a value that is as close or closer to $2^{\epsilon_q - \epsilon_p}$ than to 0, $2^{\epsilon_q - \epsilon_p}$ is added to the coefficient. Then, the operation effectively discards the $\epsilon_q - \epsilon_p$ least significant bits of each coefficient of (the entries of) $\mathbf{A} \cdot \mathbf{s}$, after which the result is assigned to $\mathbf{b}$. Subsequently, $\lfloor \mathbf{b} \rfloor_{p \to q}$ reintroduces these discarded bits, albeit all with value 0; the result of this operation is assigned to $\mathbf{b}_q$. As such, each coefficient of (the entries of) $\mathbf{b}_q$ is effectively obtained by, starting from the corresponding coefficient of $\mathbf{A} \cdot \mathbf{s}$, setting the $\epsilon_q - \epsilon_p$ least significant bits to 0 and, depending on the original value of these $\epsilon_q - \epsilon_p$ least significant bits, adding $2^{\epsilon_q - \epsilon_p}$. Indeed, the former induces a decrease of each coefficient's value by exactly the amount corresponding to its original $\epsilon_q - \epsilon_p$ least significant bits; in contrast, the latter potentially causes an increase of this value by $2^{\epsilon_q - \epsilon_p}$. More precisely, if the $\epsilon_q - \epsilon_p$ least significant bits of a coefficient of (an entry of) $\mathbf{A} \cdot \mathbf{s}$ correspond to a value $x \in [0, 2^{\epsilon_q - \epsilon_p - 1})$, then nothing is added to this coefficient; consequently, in this instance, the

[14]A common property of the considered correctness-related ranges is that they are centered around zero. To emphasize this property, the ranges are denoted with opposite boundaries, e.g., $(-a, a]$, implying that the lower bound is negative. Nevertheless, the elements contained within these ranges are still elements of $\mathbb{Z}_q$. As such, whenever convenient or desired, we might use the (positive) canonical, i.e., reduced modulo q, representatives that are congruent to these negative elements (instead of the negative elements themselves).

resulting coefficient of $\mathbf{b}_q$ is exactly x less than its counterpart from $\mathbf{A}\cdot\mathbf{s}$. Alternatively, if the $\epsilon_q - \epsilon_p$ least significant bits of a coefficient of (an entry of) $\mathbf{A}\cdot\mathbf{s}$ evaluate to a value $x \in [2^{\epsilon_q-\epsilon_p-1}, 2^{\epsilon_q-\epsilon_p})$, then $2^{\epsilon_q-\epsilon_p}$ is added to this coefficient; hence, in this case, the resulting coefficient of $\mathbf{b}_q$ is $2^{\epsilon_q-\epsilon_p} - x$ greater than its counterpart from $\mathbf{A}\cdot\mathbf{s}$. Naturally, since $x \in [2^{\epsilon_q-\epsilon_p-1}, 2^{\epsilon_q-\epsilon_p})$, it follows that $2^{\epsilon_q-\epsilon_p} - x \in (0, 2^{\epsilon_q-\epsilon_p-1}]$. At last, combining all of these observations, we conclude that the difference between the corresponding coefficients of (the entries of) $\mathbf{b}_q$ and $\mathbf{A}\cdot\mathbf{s}$, and hence each coefficient of (the entries of) $\mathbf{err}_{\mathbf{b}_q}$, lies in $(-2^{\epsilon_q-\epsilon_p-1}, 2^{\epsilon_q-\epsilon_p-1}] = (-\frac{q}{2\cdot p}, \frac{q}{2\cdot p}]$.

The second error term is analogous to the previous one, except that it relates to $\mathbf{A}^T\cdot\mathbf{s}'$ and $\mathbf{b}'_q$ instead of $\mathbf{A}\cdot\mathbf{s}$ and $\mathbf{b}_q$. Specifically, the definition of this error term is given below.

$$\mathbf{err}_{\mathbf{b}'_q} = \mathbf{b}'_q - \mathbf{A}^T\cdot\mathbf{s}' = \lfloor\lfloor\mathbf{A}^T\cdot\mathbf{s}' + \mathbf{h}\rfloor_{q\to p}\rfloor_{p\to q} - \mathbf{A}^T\cdot\mathbf{s}'$$

Due to the vast similarity with the preceding error term, this error term possesses the same properties for similar reasons. That is, following the same line of reasoning as above, $\mathbf{err}_{\mathbf{b}'_q}$ is an element of $R_q^{l\times 1}$; moreover, each coefficient of its entries lies in the discrete range $(-\frac{q}{2\cdot p}, \frac{q}{2\cdot p}]$, centered around zero.

The final error term captures the error related to $v' + \lfloor m\rfloor_{2\to q}$ and c_{mq}; this error term is defined below.

$$\mathrm{err}_{c_{mq}} = c_{mq} - (v' + \lfloor m\rfloor_{2\to q}) + \frac{q}{4\cdot t} = \lfloor\lfloor v' + \lfloor m\rfloor_{2\to q}\rfloor_{q\to 2\cdot t}\rfloor_{2\cdot t\to q} - (v' + \lfloor m\rfloor_{2\to q}) + \frac{q}{4\cdot t}$$

In contrast to the other error terms, $\mathrm{err}_{c_{mq}}$ is an element of R_q rather than of $R_q^{l\times 1}$. Moreover, $\mathrm{err}_{c_{mq}}$ is the only error term which adds a constant that is not present in the related modular scaling and flooring operations. The rational behind this is that the added constant, i.e., $\frac{q}{4\cdot t} \in R_q$, ensures that the coefficient range of $\mathrm{err}_{c_{mq}}$ is centered around zero, which is a property that is used in the exhaustive correctness computation performed by Saber's script. Without the addition of $\frac{q}{4\cdot t}$, this coefficient range is not centered around zero because the $\frac{q}{p}\cdot h_1$ term does not warrant an equivalence between the $\lfloor\cdot\rfloor_{q\to 2\cdot t}$ and $\lfloor\cdot\rceil_{q\to 2\cdot t}$ operators. Indeed, as can be extracted from the preceding discussion, the coefficient ranges for the previous two error terms are centered around zero due to the fact that $\lfloor\mathbf{A}\cdot\mathbf{s} + \mathbf{h}\rfloor_{q\to p}$ and $\lfloor\mathbf{A}^T\cdot\mathbf{s}' + \mathbf{h}\rfloor_{q\to p}$ are equivalent to $\lfloor\mathbf{A}\cdot\mathbf{s}\rceil_{q\to p}$ and $\lfloor\mathbf{A}^T\cdot\mathbf{s}'\rceil_{q\to p}$, respectively. Nevertheless, to see why the addition of $\frac{q}{4\cdot t}$ compensates for this lack of equivalence, consider the error term without this addition, i.e., $c_{mq} - (v' + \lfloor m\rfloor_{2\to q})$. Then, because $c_{mq} = \lfloor c_m\rfloor_{2\cdot t\to q} = \lfloor\lfloor v' + \lfloor m\rfloor_{2\to q}\rfloor_{q\to 2\cdot t}\rfloor_{2\cdot t\to q}$, the computation of c_{mq} commences by effectively discarding the $\frac{q}{2\cdot t} = \epsilon_q - \epsilon_t - 1$ least significant bits of each coefficient of $v' + \lfloor m\rfloor_{2\to q}$, the result of which is assigned to c_m. Subsequently, $\lfloor c_m\rfloor_{2\cdot t\to q}$ essentially reinserts these $\epsilon_q - \epsilon_t - 1$ bits, although with value 0, after which the result is assigned to c_{mq}. As such, each coefficient of c_{mq} equals its counterpart from $v' + \lfloor m\rfloor_{2\to q}$ with respect to the $\epsilon_q - (\epsilon_q - \epsilon_t - 1) = \epsilon_t + 1$ most significant bits. Nonetheless, for a coefficient of $v' + \lfloor m\rfloor_{2\to q}$, the $\epsilon_q - \epsilon_t - 1$ least significant bits can evaluate to any value $x \in [0, 2^{\epsilon_q-\epsilon_t-1})$; contrarily, for a coefficient of c_{mq}, the $\epsilon_q - \epsilon_t - 1$ least significant bits invariably evaluate to 0. Hence, each coefficient of $c_{mq} - (v' + \lfloor m\rfloor_{2\to q})$ is computed as $0 - x$, lying in the range $(-2^{\epsilon_q-\epsilon_t-1}, 0]$. Evidently, this range is not centered around zero. However, adding the term $\frac{q}{4\cdot t}$ to $c_{mq} - (v' + \lfloor m\rfloor_{2\to q})$, producing the original error term $\mathrm{err}_{c_{mq}} = c_{mq} - (v' + \lfloor m\rfloor_{2\to q}) + \frac{q}{4\cdot t}$, changes the computation of each coefficient to $0 - x + \frac{q}{4\cdot t}$. Following, since $0 - x \in (-2^{\epsilon_q-\epsilon_t-1}, 0]$, the result of this latter computation lies in $(-2^{\epsilon_q-\epsilon_t-2}, 2^{\epsilon_q-\epsilon_t-2}] = (-\frac{q}{4\cdot t}, \frac{q}{4\cdot t}]$. Consequently, we can conclude that the coefficients of $\mathrm{err}_{c_{mq}}$ lie in $(-\frac{q}{4\cdot t}, \frac{q}{4\cdot t}]$, centered around zero.

Utilizing the above-introduced error terms, we presently derive the error expression. Recall the considered context and objective: given some $m \in \mathcal{M}$, verify whether, after sequentially executing $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Saber.KeyGenA}()$, $c \leftarrow \mathsf{Saber.EncA}(\mathsf{pk}, m)$, and $m' \leftarrow \mathsf{Saber.DecA}(\mathsf{sk}, c)$, m' is equal to m, i.e., $m' = m$. Performing this sequential execution in accordance with the specifications of these algorithms, we deduce the following for $v - c_{mq} + \frac{q}{p}\cdot h_2$. Indeed, this is the value that, after modular scaling and flooring, is assigned to m' by Saber.DecA; that is, $m' = \lfloor v - c_{mq} + \frac{q}{p} \cdot h_2 \rfloor_{q \to 2}$.

$$
\begin{aligned}
v - c_{mq} + \frac{q}{p} \cdot h_2 &= v - (\mathrm{err}_{c_{mq}} + v' + \lfloor m \rfloor_{2\to q} - \frac{q}{4 \cdot t}) + \frac{q}{p} \cdot h_2 \\
&= (\mathbf{b}_q'^T \cdot \mathbf{s} + \frac{q}{p} \cdot h_1) - (\mathrm{err}_{c_{mq}} + (\mathbf{b}_q^T \cdot \mathbf{s}' + \frac{q}{p} \cdot h_1) + \lfloor m \rfloor_{2\to q} - \frac{q}{4 \cdot t}) + \frac{q}{p} \cdot h_2 \\
&= \mathbf{b}_q'^T \cdot \mathbf{s} - \mathbf{b}_q^T \cdot \mathbf{s}' - \mathrm{err}_{c_{mq}} - \lfloor m \rfloor_{2\to q} + \frac{q}{4 \cdot t} + \frac{q}{p} \cdot h_2 \\
&= (\mathbf{A}^T \cdot \mathbf{s}' + \mathbf{err}_{\mathbf{b}_q'})^T \cdot \mathbf{s} - (\mathbf{A} \cdot \mathbf{s} + \mathbf{err}_{\mathbf{b}_q})^T \cdot \mathbf{s}' - \mathrm{err}_{c_{mq}} - \lfloor m \rfloor_{2\to q} + \frac{q}{4 \cdot t} + \frac{q}{p} \cdot h_2 \\
&= ((\mathbf{A}^T \cdot \mathbf{s}')^T + \mathbf{err}_{\mathbf{b}_q'}^T) \cdot \mathbf{s} - ((\mathbf{A} \cdot \mathbf{s})^T + \mathbf{err}_{\mathbf{b}_q}^T) \cdot \mathbf{s}' - \mathrm{err}_{c_{mq}} - \lfloor m \rfloor_{2\to q} + \frac{q}{4 \cdot t} + \frac{q}{p} \cdot h_2 \\
&= (\mathbf{s}'^T \cdot \mathbf{A} + \mathbf{err}_{\mathbf{b}_q'}^T) \cdot \mathbf{s} - (\mathbf{s}^T \cdot \mathbf{A}^T + \mathbf{err}_{\mathbf{b}_q}^T) \cdot \mathbf{s}' - \mathrm{err}_{c_{mq}} - \lfloor m \rfloor_{2\to q} + \frac{q}{4 \cdot t} + \frac{q}{p} \cdot h_2 \\
&= \mathbf{s}'^T \cdot \mathbf{A} \cdot \mathbf{s} + \mathbf{err}_{\mathbf{b}_q'}^T \cdot \mathbf{s} - (\mathbf{s}^T \cdot \mathbf{A}^T \cdot \mathbf{s}' + \mathbf{err}_{\mathbf{b}_q}^T \cdot \mathbf{s}') - \mathrm{err}_{c_{mq}} - \lfloor m \rfloor_{2\to q} + \frac{q}{4 \cdot t} + \frac{q}{p} \cdot h_2 \\
&= \mathbf{err}_{\mathbf{b}_q'}^T \cdot \mathbf{s} - \mathbf{err}_{\mathbf{b}_q}^T \cdot \mathbf{s}' - \mathrm{err}_{c_{mq}} - \lfloor m \rfloor_{2\to q} + \frac{q}{4 \cdot t} + \frac{q}{p} \cdot h_2 \\
&= -\lfloor m \rfloor_{2\to q} + \mathbf{err}_{\mathbf{b}_q'}^T \cdot \mathbf{s} - \mathbf{err}_{\mathbf{b}_q}^T \cdot \mathbf{s}' - \mathrm{err}_{c_{mq}} + \frac{q}{4 \cdot t} + \frac{q}{p} \cdot (\frac{p}{4} - \frac{p}{4 \cdot t}) \\
&= -\lfloor m \rfloor_{2\to q} + \mathbf{err}_{\mathbf{b}_q'}^T \cdot \mathbf{s} - \mathbf{err}_{\mathbf{b}_q}^T \cdot \mathbf{s}' - \mathrm{err}_{c_{mq}} + \frac{q}{4} \\
&= \lfloor m \rfloor_{2\to q} + \frac{q}{4} + \mathbf{err}_{\mathbf{b}_q'}^T \cdot \mathbf{s} - \mathbf{err}_{\mathbf{b}_q}^T \cdot \mathbf{s}' - \mathrm{err}_{c_{mq}}
\end{aligned}
$$

In this derivation, most equalities follow from trivial substitutions, reorderings, and simplifications; nevertheless, a few exceptions require the use of some more intricate properties and reasoning. Specifically, the fifth and seventh equality respectively follow from the distributivity property of the transpose and matrix multiplication over matrix addition. In between these equalities, the sixth equality holds due to the relation that exists between the transpose and matrix multiplication. More precisely, for all matrices $\mathbf{X}$ and $\mathbf{Y}$ such that $\mathbf{X} \cdot \mathbf{Y}$ is well-defined, we have $(\mathbf{X} \cdot \mathbf{Y})^T = \mathbf{Y}^T \cdot \mathbf{X}^T$; naturally, since vectors are matrices with a single dimension equal to 1, this relation also applies when one or both of $\mathbf{X}$ and $\mathbf{Y}$ are vectors. Additionally, this property extends to an arbitrary number of matrices which, in combination with the involutive property of the transpose, allows for the derivation of the eighth equality. Namely, as a consequence of these properties, the terms $\mathbf{s}'^T \cdot \mathbf{A} \cdot \mathbf{s}$ and $\mathbf{s}^T \cdot \mathbf{A}^T \cdot \mathbf{s}'$ are equal and, as such, cancel each other out. More specifically, the equality between these terms can be deduced as follows.

$$
\begin{aligned}
\mathbf{s}'^T \cdot \mathbf{A} \cdot \mathbf{s} &= ((\mathbf{s}'^T \cdot \mathbf{A} \cdot \mathbf{s})^T)^T \\
&= (\mathbf{s}^T \cdot \mathbf{A}^T \cdot \mathbf{s}')^T \\
&= \mathbf{s}^T \cdot \mathbf{A}^T \cdot \mathbf{s}'
\end{aligned}
$$

Here, since $\mathbf{s}^T \cdot \mathbf{A}^T \cdot \mathbf{s}'$ is an element of R_q and, hence, no vector or matrix, the transpose effectively reduces to the identity function. Lastly, the final equality in the preceding derivation follows from the fact that $\lfloor m \rfloor_{2\to q} = -\lfloor m \rfloor_{2\to q}$. In particular, because each coefficient of m has value 0 or 1, each coefficient of its scaled counterpart $\lfloor m \rfloor_{2\to q}$ has value $0 \cdot \frac{q}{2} = 0$ or $1 \cdot \frac{q}{2} = \frac{q}{2}$; in turn, each coefficient of $-\lfloor m \rfloor_{2\to q}$ has value $-0 = 0$ or $-\frac{q}{2}$. However, since these latter coefficients are elements of $\mathbb{Z}_q$, $-\frac{q}{2}$ is congruent to $q - \frac{q}{2} = \frac{q}{2}$; consequently, each coefficient of $-\lfloor m \rfloor_{2\to q}$ is congruent to its counterpart from $\lfloor m \rfloor_{2\to q}$. Concluding, $\lfloor m \rfloor_{2\to q} = -\lfloor m \rfloor_{2\to q}$, from which the last equality follows.

Additionally considering the modular scaling and flooring operation that Saber.DecA applies to $v - c_{mq} + \frac{q}{p} \cdot h_2$, we utilize the above-derived expression as illustrated below to determine the final value Saber.DecA assigns to m'.

$$\begin{aligned} m' &= \lfloor v - c_{mq} + \frac{q}{p} \cdot h_2 \rfloor_{q \to 2} \\ &= \lfloor \lfloor m \rfloor_{2 \to q} + \frac{q}{4} + \mathbf{err}_{\mathbf{b}'_q}^T \cdot \mathbf{s} - \mathbf{err}_{\mathbf{b}_q}^T \cdot \mathbf{s}' - \mathrm{err}_{c_{mq}} \rfloor_{q \to 2} \\ &= m + \lfloor \frac{q}{4} + \mathbf{err}_{\mathbf{b}'_q}^T \cdot \mathbf{s} - \mathbf{err}_{\mathbf{b}_q}^T \cdot \mathbf{s}' - \mathrm{err}_{c_{mq}} \rfloor_{q \to 2} \end{aligned}$$

Although the first two equalities in this derivation follow from trivial substitutions, the final equality is contingent on slightly more complex reasoning; as such, we explicitly demonstrate the validity of the latter equality. Specifically, let $a_{\epsilon_q - 1} \dots a_0$ and b_0 denote the binary representations of a coefficient of $\frac{q}{4} + \mathbf{err}_{\mathbf{b}'_q}^T \cdot \mathbf{s} - \mathbf{err}_{\mathbf{b}_q}^T \cdot \mathbf{s}' - \mathrm{err}_{c_{mq}}$ and the corresponding coefficient of m, respectively. Then, the following derivation shows the validity of the third equality at the coefficient-level.

$$\begin{aligned} \lfloor \lfloor b_0 \rfloor_{2 \to q} + a_{\epsilon_q - 1} \dots a_0 \rfloor_{q \to 2} &= \lfloor b_0 0^{\epsilon_q - 1} + a_{\epsilon_q - 1} \dots a_0 \rfloor_{q \to 2} \\ &= \lfloor (a_{\epsilon_q - 1} + b_0) \dots a_0 \rfloor_{q \to 2} \\ &= (a_{\epsilon_q - 1} + b_0) \\ &= b_0 + a_{\epsilon_q - 1} \\ &= b_0 + \lfloor a_{\epsilon_q - 1} . a_{\epsilon_q - 2} \dots a_0 \rfloor \\ &= b_0 + (\lfloor a_{\epsilon_q - 1} . a_{\epsilon_q - 2} \dots a_0 \rfloor \bmod 2) \\ &= b_0 + (\left\lfloor \frac{2}{q} \cdot a_{\epsilon_q - 1} a_{\epsilon_q - 2} \dots a_0 \right\rfloor \bmod 2) \\ &= b_0 + \lfloor a_{\epsilon_q - 1} a_{\epsilon_q - 2} \dots a_0 \rfloor_{q \to 2} \end{aligned}$$

As in previous derivations, $(a_{\epsilon_q - 1} + b_0)$ represents the bit value resulting from the addition of the bits $a_{\epsilon_q - 1}$ and b_0 (modulo 2); moreover, since the addition of $b_0 0^{\epsilon_q - 1}$ and $a_{\epsilon_q - 1} \dots a_0$ is carried out in $\mathbb{Z}_q$, the implicit reduction modulo q effectively removes the $(\epsilon_q + 1)$-th bit that may be produced through a potential carry. Furthermore, due to the fact that each equality collates elements from $\mathbb{Z}_2$, the reduction modulo 2 of these elements is implied; indeed, this ratifies equalities such as $(a_{\epsilon_q - 1} + b_0) = b_0 + a_{\epsilon_q - 1}$. Lastly, because b_0 and $a_{\epsilon_q - 1} a_{\epsilon_q - 2} \dots a_0$ respectively represent any coefficient of m and the corresponding coefficient of $\frac{q}{4} + \mathbf{err}_{\mathbf{b}'_q}^T \cdot \mathbf{s} - \mathbf{err}_{\mathbf{b}_q}^T \cdot \mathbf{s}' - \mathrm{err}_{c_{mq}}$, lifting the above result to R_2 proves the validity of the previously obtained $m + \lfloor \frac{q}{4} + \mathbf{err}_{\mathbf{b}'_q}^T \cdot \mathbf{s} - \mathbf{err}_{\mathbf{b}_q}^T \cdot \mathbf{s}' - \mathrm{err}_{c_{mq}} \rfloor_{q \to 2}$, as desired.

Employing the above-derived equality for m', i.e., $m' = m + \lfloor \frac{q}{4} + \mathbf{err}_{\mathbf{b}'_q}^T \cdot \mathbf{s} - \mathbf{err}_{\mathbf{b}_q}^T \cdot \mathbf{s}' - \mathrm{err}_{c_{mq}} \rfloor_{q \to 2}$, we can directly infer that $m' = m$ if and only if $\lfloor \frac{q}{4} + \mathbf{err}_{\mathbf{b}'_q}^T \cdot \mathbf{s} - \mathbf{err}_{\mathbf{b}_q}^T \cdot \mathbf{s}' - \mathrm{err}_{c_{mq}} \rfloor_{q \to 2} = 0$. Evaluating the modular scaling and flooring operation of the latter, it follows that $\lfloor \frac{q}{4} + \mathbf{err}_{\mathbf{b}'_q}^T \cdot \mathbf{s} - \mathbf{err}_{\mathbf{b}_q}^T \cdot \mathbf{s}' - \mathrm{err}_{c_{mq}} \rfloor_{q \to 2} = 0$ if and only if the ϵ_q-th bit of each coefficient of $\frac{q}{4} + \mathbf{err}_{\mathbf{b}'_q}^T \cdot \mathbf{s} - \mathbf{err}_{\mathbf{b}_q}^T \cdot \mathbf{s}' - \mathrm{err}_{c_{mq}}$ equals 0. In turn, since each coefficient consists of ϵ_q bits, the ϵ_q-th bit of a coefficient can be 0 if and only if this coefficient lies in the discrete range $[0, 2^{\epsilon_q - 1}) = [0, \frac{q}{2})$. Then, subtracting $\frac{q}{4}$ from $\frac{q}{4} + \mathbf{err}_{\mathbf{b}'_q}^T \cdot \mathbf{s} - \mathbf{err}_{\mathbf{b}_q}^T \cdot \mathbf{s}' - \mathrm{err}_{c_{mq}}$, we obtain the error expression for which Saber's script exhaustively computes the distribution, i.e., $\mathbf{err}_{\mathbf{b}'_q}^T \cdot \mathbf{s} - \mathbf{err}_{\mathbf{b}_q}^T \cdot \mathbf{s}' - \mathrm{err}_{c_{mq}}$; accordingly, the aforementioned coefficient range also shifts in the negative direction by a constant $\frac{q}{4}$. Lastly, we conclude that $m' = m$ if and only if each coefficient of $\mathbf{err}_{\mathbf{b}'_q}^T \cdot \mathbf{s} - \mathbf{err}_{\mathbf{b}_q}^T \cdot \mathbf{s}' - \mathrm{err}_{c_{mq}}$ lies in the discrete range $[0 - \frac{q}{4}, \frac{q}{2} - \frac{q}{4}) = [-\frac{q}{4}, \frac{q}{4})$.

As a last endeavor preceding the conclusion of this discussion, we prove that the error term $\mathrm{err}_{c_{mq}}$ is independent of the message m. As a result, since the other error terms do not contain m, it follows that the complete error expression is independent of the message as well. Initiating the proof, the imminent derivation performs a trivial substitution to acquire a convenient representation of

$\mathrm{err}_{c_{mq}}$.

$$\begin{aligned}\mathrm{err}_{c_{mq}} &= c_{mq} - (v' + \lfloor m \rfloor_{2\to q}) + \frac{q}{4\cdot t}\\ &= \lfloor \lfloor v' + \lfloor m \rfloor_{2\to q} \rfloor_{q\to 2\cdot t} \rfloor_{2\cdot t \to q} - (v' + \lfloor m \rfloor_{2\to q}) + \frac{q}{4\cdot t}\end{aligned}$$

Subsequently, we extract $\lfloor m \rfloor_{2\to q}$ from the modular scaling and flooring operations. In particular, respectively denoting the binary representations of a coefficient of v' and the corresponding coefficient of m by $a_{\epsilon_q - 1} \dots a_0$ and b_0, the ensuing derivation illustrates this extraction on the coefficient-level. Here, because the right part of the formula remains untouched, $(v' + \lfloor m \rfloor_{2\to q}) + \frac{q}{4\cdot t}$ is replaced by dots.

$$\begin{aligned}\lfloor \lfloor a_{\epsilon_q-1} \dots a_0 + \lfloor b_0 \rfloor_{2\to q} \rfloor_{q\to 2\cdot t} \rfloor_{2\cdot t\to q} - \dots &= \lfloor \lfloor a_{\epsilon_q-1} \dots a_0 + b_0 0^{\epsilon_q - 1} \rfloor_{q\to 2\cdot t} \rfloor_{2\cdot t\to q} - \dots\\ &= \lfloor \lfloor (a_{\epsilon_q-1} + b_0) \dots a_0 \rfloor_{q\to 2\cdot t} \rfloor_{2\cdot t\to q} - \dots\\ &= \lfloor (a_{\epsilon_q-1} + b_0) \dots a_{\epsilon_q-\epsilon_t-1} \rfloor_{2\cdot t\to q} - \dots\\ &= (a_{\epsilon_q-1} + b_0) \dots a_{\epsilon_q-\epsilon_t-1} 0^{\epsilon_q-\epsilon_t-1} - \dots\\ &= b_0 0^{\epsilon_q-1} + a_{\epsilon_q-1} \dots a_{\epsilon_q-\epsilon_t-1} 0^{\epsilon_q-\epsilon_t-1} - \dots\\ &= \lfloor b_0 \rfloor_{2\to q} + \lfloor a_{\epsilon_q-1} \dots a_{\epsilon_q-\epsilon_t-1} \rfloor_{2\cdot t\to q} - \dots\\ &= \lfloor b_0 \rfloor_{2\to q} + \lfloor \lfloor a_{\epsilon_q-1} \dots a_{\epsilon_q-\epsilon_t-1}.a_{\epsilon_q-\epsilon_t-2} \dots a_0 \rfloor \rfloor_{2\cdot t\to q} - \dots\\ &= \lfloor b_0 \rfloor_{2\to q} + \lfloor \lfloor a_{\epsilon_q-1} \dots a_0 \rfloor_{q\to 2\cdot t} \rfloor_{2\cdot t\to q} - \dots\end{aligned}$$

As before, because the above holds for arbitrary (corresponding) coefficients of v' and m, lifting this result to R_2 gives $(\lfloor m \rfloor_{2\to q} + \lfloor \lfloor v' \rfloor_{q\to 2\cdot t} \rfloor_{2\cdot t\to q}) - (v' + \lfloor m \rfloor_{2\to q}) + \frac{q}{4\cdot t}$. Naturally, this straightforwardly reduces to $\lfloor \lfloor v' \rfloor_{q\to 2\cdot t} \rfloor_{2\cdot t\to q} - v' + \frac{q}{4\cdot t}$, an expression that does not depend on m.

Finally, utilizing the preceding observations and results, we can reach the desired conclusion. Namely, according to the previously obtained results, given some $m \in \mathcal{M}$, after sequentially executing $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Saber.KeyGenA}()$, $c \leftarrow \mathsf{Saber.EncA}(\mathsf{pk}, m)$, and $m' \leftarrow \mathsf{Saber.DecA}(\mathsf{sk}, c)$, verifying whether $m' = m$ is equivalent to verifying whether all coefficients of the error expression $\mathbf{err}_{\mathbf{b}'_q}^T \cdot \mathbf{s} - \mathbf{err}_{\mathbf{b}_q}^T \cdot \mathbf{s}' - \mathrm{err}_{c_{mq}}$ lie in the discrete range $[-\frac{q}{4}, \frac{q}{4})$. Furthermore, since this error expression is completely independent of m, the particular choice for m can not affect the outcome of this latter verification effort; in turn, due to the equivalence, this also holds for the former verification effort, i.e., the verification of $m' = m$ after the above-mentioned sequential execution. In fact, completely unfolding the error expression, we see that, excluding any constants, it solely depends on $\mathbf{A}$, $\mathbf{s}$, and $\mathbf{s}'$ produced as in $\mathsf{Saber.KeyGenA}$, $\mathsf{Saber.KeyGenA}$, and $\mathsf{Saber.EncA}$, respectively. As such, we can formalize the corresponding verification effort as a probabilistic program; Figure 3.15 defines this probabilistic program. Here, err_expression$(\mathbf{A}, \mathbf{s}, \mathbf{s}')$ represents the error expression $\mathbf{err}_{\mathbf{b}'_q}^T \cdot \mathbf{s} - \mathbf{err}_{\mathbf{b}_q}^T \cdot \mathbf{s}' - \mathrm{err}_{c_{mq}}$, accordingly using the provided arguments as the values for $\mathbf{A}$, $\mathbf{s}$, and $\mathbf{s}'$; moreover, for $x \in R_q$, coeffs_in_correctness_rng(x) denotes the predicate that evaluates to true if and only if each of x's coefficients lies in $[-\frac{q}{4}, \frac{q}{4})$.

```
PProg^COR
1 : $\mathrm{seed}_{\mathbf{A}} \leftarrow\$ \mathcal{U}(\{0,1\}^{256})$
2 : $\mathbf{A} \leftarrow \mathsf{gen}(\mathrm{seed}_{\mathbf{A}})$
3 : $\mathbf{s} \leftarrow\$ \mathcal{U}(R_q^{l\times 1})$
4 : $\mathbf{s}' \leftarrow\$ \mathcal{U}(R_q^{l\times 1})$
5 : return coeffs_in_correctness_rng(err_expression($\mathbf{A}, \mathbf{s}, \mathbf{s}'$))
```

Figure 3.15: Probabilistic Program Formalizing Correctness Probability Based on Error Expression

Recalling that both $\text{PProg}^{\text{STDCOR}}_{\text{Saber.PKEA}}$ and $\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{Saber.PKEA}}$ formalize the verification of $m' = m$ after the sequential execution of Saber.PKEA's algorithms, we notice that both of these programs are equivalent to $\text{PProg}^{\text{COR}}$. In particular, the independence of m in this verification effort nullifies the only difference between $\text{PProg}^{\text{STDCOR}}_{\text{Saber.PKEA}}$ and $\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{Saber.PKEA}}$, i.e., the selection of m, and enables the (equivalence-preserving) transformations of both programs to $\text{PProg}^{\text{COR}}$. From these equivalences, it follows that for any $m \in \mathcal{M}$, $\Pr\Big[\text{PProg}^{\text{STDCOR}}_{\text{Saber.PKEA}}(m) = 1\Big]$ is equal to $\Pr\Big[\text{PProg}^{\text{COR}} = 1\Big]$; similarly, for any valid adversary $\mathcal{A}$, $\Pr\Big[\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{Saber.PKEA}} = 1\Big]$ equals $\Pr\Big[\text{PProg}^{\text{COR}} = 1\Big]$. Then, since $\text{PProg}^{\text{COR}}$ does not depend on m, $\Pr\Big[\text{PProg}^{\text{COR}} = 1\Big]$ remains constant for any choice of message (or adversary that produces such a message). Consequently, we can deduce that there exists a $1 - \delta \in [0, 1]$ such that for any message $m \in \mathcal{M}$ and any valid adversary $\mathcal{A}$, the following holds.

$$\Pr\Big[\text{PProg}^{\text{COR}} = 1\Big] = \Pr\Big[\text{PProg}^{\text{STDCOR}}_{\text{Saber.PKEA}}(m) = 1\Big] = \Pr\Big[\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{Saber.PKEA}} = 1\Big] = 1 - \delta$$

Thus, by computing $\Pr\Big[\text{PProg}^{\text{COR}} = 1\Big]$, we obtain the correctness of Saber.PKEA with respect to both considered definitions; naturally, due to the equivalence between Saber.PKEA and Saber.PKE, this correctness result is identical for Saber.PKE. Certainly, given a particular parameter set, this is precisely the probability that Saber's script exhaustively computes, albeit under the assumption that matrix $\mathbf{A}$ is uniformly distributed; alternatively stated, the script assumes gen is a random oracle[15]. With this assumption, $\text{PProg}^{\text{COR}}$ essentially becomes $\text{PProg}^{\delta\text{COR}}$, the probabilistic program specified in Figure 3.16. Furthermore, in the above sequence of equalities, the initial probability should therefore technically be $\Pr\Big[\text{PProg}^{\delta\text{COR}} = 1\Big]$ instead of $\Pr\Big[\text{PProg}^{\text{COR}} = 1\Big]$. Combining these final observations, we derive the following concluding sequence of equalities, completing the correctness analysis of Saber.PKE. Indeed, as before, these equalities hold for any $m \in \mathcal{M}$ and any valid adversary $\mathcal{A}$.

$$\Pr\Big[\text{PProg}^{\delta\text{COR}} = 1\Big] = \Pr\Big[\text{PProg}^{\text{STDCOR}}_{\text{Saber.PKE}}(m) = 1\Big] = \Pr\Big[\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{Saber.PKE}} = 1\Big] = 1 - \delta$$

```
$\text{PProg}^{\delta\text{COR}}$
1 :  $\mathbf{A} \leftarrow \mathcal{U}(R_q^{l \times l})$
2 :  $\mathbf{s} \leftarrow\$ \ \mathcal{U}(R_q^{l \times 1})$
3 :  $\mathbf{s}' \leftarrow\$ \ \mathcal{U}(R_q^{l \times 1})$
4 :  return coeffs_in_correctness_rng(err_expression($\mathbf{A}, \mathbf{s}, \mathbf{s}'$))
```

Figure 3.16: Probabilistic Program Formalizing Correctness Probability Computed by Saber's Script

Implications and Concrete Correctness Values

As mentioned in the beginning of the correctness analysis, determining the correctness of Saber.PKE is imperative for two primary purposes. First, as for any PKE scheme, the (sufficient) correctness of Saber.PKE validates that the scheme's algorithms (adequately) satisfy the desired relation with respect to the valid inputs; in essence, this relation expresses that encrypting and, subsequently, decrypting any valid message through the scheme's algorithms returns this message with sufficient

[15]This suggests that Saber's script merely approximates the actual correctness value. Nevertheless, if gen is adequately instantiated, i.e., its output distribution (closely) resembles the uniform distribution over $R_q^{l \times l}$, this approximation is (almost) accurate.

probability. Second, due to (the variant of) the FO transformation used to obtain Saber.KEM from Saber.PKE, the security and correctness properties of Saber.KEM are predicated on the correctness property of Saber.PKE; in particular, the correctness of Saber.KEM is identical to the correctness of Saber.PKE [14, 25]. Therefore, determining the correctness of Saber.PKE is also necessary to evaluate the properties of Saber.KEM.

Although the above-mentioned purposes technically employ slightly different correctness definitions, these definitions are equivalent in the context of Saber.PKE. As such, considering a particular parameter set and a proper instantiation of `gen`, Saber's script (almost) accurately computes the correctness of Saber.PKE with respect to either definition and, hence, purpose. Certainly, this implies that, for a certain parameter set and an adequate instantiation of `gen`, the correctness value computed by Saber's script can directly be employed to evaluate Saber.KEM's security and correctness properties; most notably, this correctness value additionally (approximately) denotes Saber.KEM's correctness.

Finally, to give an impression of the envisioned correctness of Saber.PKE and Saber.KEM, we provide the concrete correctness values induced by the parameter sets advertised in the original paper; these sets are denominated LightSaber, (regular) Saber, and FireSaber and instantiate the parameters as follows [14].

- For LightSaber, $\epsilon_t = 2$, $\epsilon_p = 10$, $\epsilon_q = 13$, $\epsilon_n = 8$, $l = 2$, and $\mu = 10$.
- For (regular) Saber, $\epsilon_t = 3$, $\epsilon_p = 10$, $\epsilon_q = 13$, $\epsilon_n = 8$, $l = 3$, and $\mu = 8$.
- For FireSaber, $\epsilon_t = 5$, $\epsilon_p = 10$, $\epsilon_q = 13$, $\epsilon_n = 8$, $l = 4$, and $\mu = 6$.

With respect to these parameter sets, Table 3.1 presents the corresponding concrete correctness values of Saber.PKE and Saber.KEM, computed per Saber's script.

Parameter Set	**Correctness of** Saber.PKE **and** Saber.KEM
LightSaber	$1 - 2^{-120}$
(Regular) Saber	$1 - 2^{-136}$
FireSaber	$1 - 2^{-165}$

Table 3.1: Correctness of Saber.PKE and Saber.KEM for Customary Parameter Sets

Chapter 4

Formal Verification

Closely resembling the extensive and detailed discussion provided in the previous chapter, this chapter covers the formal verification effort carried out for Saber.PKE. Specifically, this chapter describes the process of formally verifying Saber.PKE, elaborating on the critical aspects, decisions, and difficulties. Although this does not encompass a comprehensive walkthrough of the code, one can independently access and examine the entire codebase in my GitHub repository, located at the following URL.

`https://github.com/MM45/Saber-Security-EasyCrypt`

Introduced in Chapter 1 and further expanded on in Chapter 2, EasyCrypt is the sole tool utilized for the formal verification of Saber.PKE. As such, the corresponding formal verification process is entirely discussed from the perspective of this tool; in particular, this discussion provides ample examples from the actual code or slightly adjusted variants thereof. In order to facilely interpret and understand the material of this discussion, acquaintance with the information on EasyCrypt presented in the initial two chapters is strongly recommended.

This chapter follows a similar structure to that of the preceding chapter. Namely, foremost, Section 4.1 covers the preliminary formalizations; more precisely, this section regards the formalization of the considered context, i.e., fundamental definitions and properties, in EasyCrypt. Indeed, these definitions and properties predominantly correspond to those introduced in Section 3.1. Afterward, Section 4.2 discusses the formalization and (formal) verification of the analyses provided in Section 3.2. Finally, to give an impression of EasyCrypt's (lemma-)proving process and its underlying concepts and mechanisms, Section 4.3 explicates the concrete proof of one of the lemmas employed in the formal verification endeavor for Saber.PKE.

4.1 Preliminaries

Before formalizing any concrete schemes, security properties, hardness assumptions, or proofs, we must formalize the setting in which they manifest themselves. Particularly, this includes introducing the necessary types, operators, constants, distributions, and axioms; certainly, in the formal verification effort performed for Saber.PKE, these artifacts primarily formalize the definitions, requirements, and concepts introduced in Section 3.1. In the ensuing, we discuss this initial endeavor.

4.1.1 Saber Parameters and Axioms

Foremost, we formalize Saber's parameters and the corresponding requirements. The rationale behind formalizing these artifacts first is that they constitute the most fundamental part of the considered context; indeed, (nearly) the entire remainder of the context is contingent on these artifacts. For instance, the utilized algebraic structures, e.g., R_q, modular scaling and flooring operators, e.g., $\lfloor \cdot \rceil_{q \to p}$, and distributions, e.g. $\beta_\mu(R_q^{l \times 1})$, are all defined with respect to the parameters.

The parameters of Saber are formalized by the integer constants defined in Listing 4.1.

```
(* Exponents *)
const eq: int.
const ep: int.
const et: int.
const en: int.

(* Moduli (q, p, and t) and Polynomial Degree (n) *)
const q: int = 2^eq.
const p: int = 2^ep.
const t: int = 2^et.
const n: int = 2^en.

(* Vector/Matrix Dimension *)
const l: int.
```

Listing 4.1: Saber's Parameters

As we can see, the moduli and polynomial degree are assigned integral power-of-two values based on their exponent, conforming to the definitions provided in Section 3.1. As an example, `eq` denotes exponent ϵ_q; accordingly, `q`, representing modulus q, is given the value `2^eq`, i.e., 2^{ϵ_q}. The remaining constant, `l`, corresponds to parameter l; that is, `l` determines the dimensions of the vectors and matrices utilized in Saber.PKE.

In Listing 4.1, the parameter of Saber's centered binomial distribution, i.e., μ, is not formalized. This is because we do not explicitly consider the centered binomial distribution in this formal verification effort. Instead, we analyze a more general, abstract distribution, yielding a broader verification effort with stronger guarantees. Particularly, the guarantees of this broader verification effort are valid for any concrete distribution compatible with the more general distribution, including the centered binomial distribution employed in Saber.PKE.

Following the formalization of the parameters, we specify the associated axioms. These axioms encompass the most rudimentary restrictions by which the parameters must abide; that is, in this case, they correspond to the parameter requirements presented in Section 3.1. The concrete specifications of these axioms are shown in Listing 4.2.

```
(* Requirements/Assumptions on Parameters *)
axiom ge1_et1: 1 <= et + 1.
axiom geet2_ep: et + 2 <= ep.
axiom geep1_eq: ep + 1 <= eq.

axiom ge0_en: 0 <= en.

axiom ge1_l: 1 <= l.
```

Listing 4.2: Saber's Parameter Requirements

Here, the first three axioms concern the relation between the moduli exponents. Specifically, in order, they ensure $1 \leq \epsilon_t + 1$, $\epsilon_t + 2 \leq \epsilon_p$, and $\epsilon_p + 1 \leq \epsilon_q$; hence, in conjunction, these axioms imply the desired relation between the moduli exponents. The fourth axiom in this listing ensures that $0 \leq \epsilon_n$; as a consequence, $n = 2^{\epsilon_n}$ is an integral power-of-two. At last, the final axiom guarantees that $1 \leq l$ or, equivalently, $0 < l$; in other words, this axiom guarantees the validity of the vector and matrix dimensions. In conclusion, combined, these axioms precisely formalize the list of parameter requirements described in Section 3.1. Naturally, this is with the exception of the requirement on μ, since this parameter is not considered in this formal verification effort.

4.1.2 Types, Operators, and Distributions

Building on the most rudimentary context, i.e., the parameters and their axioms, we formalize the remaining concepts required to model Saber.PKE and its algorithms; in particular, these concepts involve the necessary algebraic structures, operators, and distributions.

Polynomial Quotient Ring Theory

As is apparent from the discussion in Chapter 3, Saber.PKE utilizes several polynomial quotient rings with a nearly identical structure, e.g., R_q, R_p, and $R_{2 \cdot t}$. As such, to prevent unnecessary, error-prone code duplication, a theory defining the general structure of such rings would be conducive. Unfortunately, at present, EasyCrypt's standard library does not provide such a theory. For this reason, we devise a theory for polynomial quotient rings of the relevant form, i.e., $K[X]/(X^n + 1)$ where K is a ring. Moreover, as a concretization of this theory, we create a separate theory for polynomial quotient rings with the exact shape of the ones employed in Saber.PKE, i.e., $\mathbb{Z}_a[X]/(X^n + 1)$ where $a \in \mathbb{N}$ and $1 < a$.

In essence, the foundation of the general polynomial quotient ring theory is predominantly based on the definitions, properties, and structure provided in preexisting theories. Specifically, we utilize the polynomial, ideal, and ring quotient theories to establish the foundation of the general polynomial quotient ring theory. Namely, first, the polynomial theory provides the representation of polynomials as well as the definitions and properties of related concepts, e.g., the degree of a polynomial. Second, the ideal theory enables the construction of the desired ideal, i.e., $\langle X^n + 1 \rangle$; furthermore, it defines the properties associated with this ideal. Lastly, the ring quotient theory describes the structure of the quotient ring induced by the constructed ideal.

Leveraging this foundation, we further develop the general polynomial quotient ring theory to include the definitions and properties regarding polynomial quotient rings of the considered form. These additional definitions and properties primarily concern the canonical, i.e., (fully) reduced modulo $X^n + 1$, representatives of the congruence classes in such rings, as well as the operators on these congruence classes.

Finally, we construct the more concrete polynomial quotient ring theory, i.e., the theory for rings of the form $\mathbb{Z}_a[X]/(X^n + 1)$, by cloning the above-discussed general theory and replacing the general coefficient ring with the ring of integers modulo a, where $a \in \mathbb{N}$ and $1 < a$; this replacement uses the predefined theory for such integer rings. Based on the uniform distribution supplied in this predefined theory, the concrete polynomial quotient ring theory specifies a uniform distribution on $\mathbb{Z}_a[X]/(X^n + 1)$. Apart from the added structure from the more specific coefficient ring, this distribution is the only additional feature of the concrete polynomial quotient ring theory relative to its general counterpart.

Types

Types are used to capture the origin of a certain value, indicating the associated structure and compatible operators. In the formal verification effort regarding Saber.PKE, types are mainly utilized to represent the necessary algebraic structures from which the considered elements originate. Nonetheless, several supplementary types are employed to denote (the domain of) abstract cryptographic artifacts such as plaintexts, ciphertexts, public keys, and secret keys.

First, exemplifying the formalization of the most prevalent algebraic structures employed in Saber.PKE, i.e., the polynomial quotient rings and their coefficient rings, Listing 4.3 considers the definition of the types corresponding to $R_q = \mathbb{Z}_q[X]/(X^n + 1)$ and its coefficient ring $\mathbb{Z}_q$.

```
1  (* Rq = Z/qZ[X] / (X^n + 1) *)
2  type Zq, Rq.
3
4  clone include PolyZ with
5    type Zmod.zmod     <- Zq,
6    type Rmod.polyXnD1 <- Rq,
7      op Zmod.p        <- q.
```

Listing 4.3: Types for Polynomial Quotient Ring R_q and Corresponding Coefficient Ring $\mathbb{Z}_q$

In this listing, `Zq` and `Rq` respectively represent $\mathbb{Z}_q$ and R_q. However, line 2 merely defines abstract types with identifiers `Zq` and `Rq`; that is, from EasyCrypt's perspective, no structure or properties are yet associated with these types. The subsequent cloning of the `PolyZ` theory remedies this deficiency. Namely, `PolyZ` corresponds to the previously discussed concrete polynomial quotient ring theory. Indeed, this suggests that the cloning process depicted in Listing 4.3 creates an instance of the concrete polynomial quotient ring theory for which `Zq` and `Rq` constitute the types of the coefficient ring elements and polynomial quotient ring elements, respectively. As a result of this cloning process, both `Zq` and `Rq` have the intended structure and properties associated with them. Although this example solely covers $\mathbb{Z}_q$ and R_q, an analogous process is carried out for each necessary coefficient ring and corresponding polynomial quotient ring; more precisely, a similar process is carried out for $\mathbb{Z}_p$ and R_p, $\mathbb{Z}_{p^2/q}$ and $R_{p^2/q}$, $\mathbb{Z}_{2 \cdot t}$ and $R_{2 \cdot t}$, and $\mathbb{Z}_2$ and R_2.

Second, based on the types for polynomial quotient rings, we formalize the algebraic structures that consider vectors and matrices over these polynomial quotient rings; indeed, as can be extracted from Chapter 3, Saber.PKE heavily relies on the use of such vectors and matrices. Specifically, Saber.PKE utilizes l-dimensional vectors over R_q and R_p, and $l \times l$-dimensional matrices over R_q. Fortunately, EasyCrypt's standard library contains the `Matrix` theory, which provides the fundamental definitions and properties regarding vectors and square matrices[1]. Among these definitions is a type for vectors and matrices of a particular dimension over some entry type. As such, instantiating this theory with dimension `l` and entry type `Rq` yields types for the l-dimensional vectors and $l \times l$-dimensional matrices over R_q, along with the corresponding structure, definitions, and properties. Naturally, instantiating this theory anew with dimension `l` and entry type `Rp` gives an analogous type for the l-dimensional vectors over R_p. Albeit slightly simplified, Listing 4.4 depicts this instantiation process. The `(*...*)` comments indicate places where the listing abstracts away from some initialization details.

```
clone Matrix as Mat_Rq with
  type R         <- Rq,
    op size      <- l.

(*...*)

type Rq_vec = Mat_Rq.vector.
type Rq_mat = Mat_Rq.Matrix.matrix.


```

[1]Indeed, at the time of writing, the `Matrix` theory exclusively considers square matrices.

```
clone Matrix as Mat_Rp with
  type R         <- Rp,
    op size      <- l.

(*...*)

type Rp_vec = Mat_Rp.vector.
```

Listing 4.4: Types for Vectors and Matrices over R_q and R_p

Contrariwise to the cloned instance of `PolyZ` presented earlier, the cloned instances of `Matrix` are renamed. Specifically, the instance with entry type `Rq` is renamed to `Mat_Rq`; the instance with entry type `Rp` is renamed to `Mat_Rp`. Subsequently, the instances' content may be accessed through their respective identifiers. Furthermore, for readability purposes, we define aliases for the relevant vector and matrix types that these instances provide.

Lastly, the types representing abstract cryptographic artifacts are shown in Listing 4.5. From top to bottom, the listed types formalize (the domain of) seeds, public keys, secret keys, plaintexts, and ciphertexts. Technically, these types are not necessary for the formal verification to succeed; however, as will become apparent throughout the remainder, they considerably simplify the formalization of the security proof.

```
(* Cryptographic Types *)
type seed.
type pkey.
type skey.
type plaintext.
type ciphertext.
```

Listing 4.5: Types Representing Abstract Cryptographic Artifacts

Operators

Utilizing the defined types, we specify the remainder of the necessary operators; notably, this remainder does not include the operators that are inherent to the utilized algebraic structures. Namely, albeit not explicitly presented here, such operators are among the artifacts (abstractly) specified within the theories that formalize the corresponding algebraic structures. Indeed, this means that the above-discussed instantiations of the `PolyZ` and `Matrix` theories with the desired types automatically concretize and provide the inherent operators of the corresponding algebraic structures for these types.

Besides the operators that are inherent to the utilized algebraic structures, Saber.PKE predominantly requires two additional categories of operators: modular reduction (and conversion) operators and modular scaling and flooring operators. Although EasyCrypt's standard library provides most of the fundamental definitions and properties necessary to formalize these operators, the exact operators employed in Saber.PKE are not predefined and, hence, must still be formalized. In addition to these two categories of operators, we employ several encoding and decoding operators related to the above-introduced cryptographic types; in fact, for each of these types (excluding the `seed` type), we define both a monomorphic and a polymorphic encoding/decoding operator pair. Here, the monomorphic operators are utilized in the formalizations of Saber.PKE and Saber.PKEA; the polymorphic operators are employed in the formalization of the game-playing security proof. The rationale behind the manner in which these operators are used is bipartite: the formal verification of the equivalence proof regarding Saber.PKE and Saber.PKEA requires explicitly referring to the concrete types of the considered algorithms' input and output, necessitating monomorphic operators; and the formal verification of the game-playing security proof warrants managing the deviating adversary interfaces, an endeavor significantly simplified by the polymorphic operators.

Exemplifying the latter part of this rationale, consider $\text{Game}^2_{\mathcal{A}}$ and $\text{Game}^3_{\mathcal{A}}$ of the security proof's game sequence (see Figure 3.8). Certainly, the values passed to the adversary in $\text{Game}^2_{\mathcal{A}}$, i.e., $(\text{seed}_{\mathbf{A}}, \mathbf{b})$ and $(c_u, \mathbf{b}')$, respectively belong to the domains $\mathcal{U}(\{0,1\}^{256}) \times R_p^{l\times 1}$ and $R_{p^2/q} \times R_p^{l\times 1}$; contrarily, in $\text{Game}^3_{\mathcal{A}}$, these values respectively belong to the domains $\mathcal{U}(\{0,1\}^{256}) \times R_q^{l\times 1}$ and $R_p \times R_p^{l\times 1}$. As such, because each domain has a distinct type in EasyCrypt, the formal verification of the game-playing security proof would require several different adversary definitions or complex and cumbersome type conversions; this is circumvented through the above-mentioned polymorphic encoding and decoding operators. Naturally, to preserve the equivalence between the IND-CPA game for Saber.PKE and the initial game in the game sequence of the security proof, the polymorphic operators are defined to be equal to their monomorphic analogs for the corresponding concrete domain and range types. Finally, a single additional operator models the `gen` function. This operator is left rather abstract, merely mapping a seed to an element of $R_q^{l\times l}$ without further properties or requirements.

Modular Reduction and Conversion Concerning the first additional category of operators, Listing 4.6 provides the formalizations of a modular reduction (and conversion) operator and its extensions. Although this listing exclusively considers operators that convert from modulo q to modulo p, the actual formal verification effort comprises more of such operators for different moduli. Furthermore, facilitating their identification, all (formalizations of the) modular reduction and conversion operators adhere to the same identifier format. Specifically, if such an operator converts a value of type `X` to a value of type `Y`, the operator is identified by `X2Y`.

```
(* Modular Reduction/Modulus Conversion *)
op Zq2Zp (z : Zq) : Zp = Zp.inzmod (Zq.asint z).

(* Extend Modular Reduction/Modulus Conversion to Polynomials *)
op Rq2Rp (y : Rq) : Rp =
  BigRp.XnD1CA.bigi predT (fun (i : int) => Zq2Zp y.[i] ** exp Rp.iX i) 0 n.

(* Extend Modular Reduction/Modulus Conversion to Polynomial Vectors *)
op Rqv2Rpv (v : Rq_vec) : Rp_vec =
  Mat_Rp.Vector.offunv (fun (i : int) => Rq2Rp v.[i]).
```

Listing 4.6: Modular Reduction and Conversion Operator with Extensions

Here, the first operator, i.e., `Zq2Zp`, is designed to convert a value from type `Zq` to type `Zp`, implicitly reducing the value modulo `p`. Particularly, it performs this reduction and conversion by interpreting the given value as an integer and, subsequently, reinterpreting it as a value of type `Zp`. Indeed, this operator formalizes the (overloaded) mod p operator that maps from $\mathbb{Z}_q$ to $\mathbb{Z}_p$, defined in Section 3.1. The second operator, i.e., `Rq2Rp`, extends the `Zq2Zp` operator to polynomials of type `Rq`. Dissecting this operator's definition, first and foremost, `BigRp.XnD1CA.bigi` denotes a summation operator for values of type `Rp`. This summation operator takes a predicate, function, and two integers as arguments. Generally, the predicate argument is used to exclude certain values from the summation. However, in this case, the predicate equals `predT`, which invariably evaluates to true; consequently, the summation does not exclude any values. The function argument defines the function from which the summation terms originate. More precisely, each summation term results from evaluating this function on a value from the range defined by the two integer arguments. In this instance, the summation terms are obtained from evaluating `(fun (i : int) => Zq2Zp y.[i] ** exp Rp.iX i)` on integers from `0` (including) up to `n` (excluding). In the definition of this function, `y.[i]` refers to the coefficient with index `i` of polynomial `y`, `exp Rp.iX i` represents the monomial X^i interpreted as an element from R_p, and `**` denotes constant multiplication, i.e., multiplication of each coefficient of the right-hand side polynomial with the left-hand side constant. As such, letting Zq2Zp, Rq2Rp, y_i and X^i respectively represent `Zq2Zp`, `Rq2Rp`, `y.[i]`

and `exp Rp.iX i`, we can derive the following equality for the `Rq2Rp` operator.

$$\mathrm{Rq2Rp}(y) = \sum_{i=0}^{n-1} \mathrm{Zq2Zp}(y_i) \cdot X^i$$

Since `Zq2Zp` solely reduces its argument modulo p, `Rq2Rp` produces the polynomial that results from reducing each of y's coefficients modulo p. As a result, `Rq2Rp` exactly corresponds to the coefficient-wise extension of the aforementioned mod p operator; that is, the extension mapping from R_q to R_p.

As a final necessary extension, the last operator in Listing 4.6, i.e., `Rqv2Rpv`, extends the `Rq2Rp` operator to elements of type `Rq_vec`, i.e., `l`-dimensional vectors with entries of type `Rq`. Specifically, `Rqv2Rpv` implements this extension through the `Mat_Rp.Vector.offunv` operator, which constructs a vector of type `Rp_vec` from a given function. In this case, this given function is `(fun (i : int) => Rq2Rp v.[i])`, where `v.[i]` refers to the entry with index `i` of vector `v`. Hence, combining this with the above discussion on `Rq2Rp`, it follows that `Rqv2Rpv` converts each entry of the argument vector from type `Rq` to `Rp`, producing a value of type `Rp_vec`; particularly, it does so by reducing all coefficients of each entry modulo `p`. Concluding, `Rqv2Rpv` precisely corresponds to the combination of the coefficient-wise and entry-wise extensions of the above-mentioned mod p operator, i.e., the combination of extensions mapping from $R_q^{l \times 1}$ to $R_p^{l \times 1}$.

Modular Scaling and Flooring For the second additional category of operators, i.e., the modular scaling and flooring operators, Listing 4.7 defines several necessary preliminary operators. Here, the `%/` operator returns the quotient resulting from the Euclidean division of the left-hand side operand by the right-hand side operand. This operator and its properties are defined in EasyCrypt's standard library.

```
(* Right and Left Bit-Shift *)
op shr (x : int, ex : int) : int = x %/ 2 ^ ex.
op shl (x : int, ex : int) : int = x * 2 ^ ex.

(* Right and Left Bit-Shift *)
op downscale (x : int, ea : int, eb : int) : int = shr x (ea - eb).
op upscale (x : int, ea : int, eb : int) : int = shl x (ea - eb).
```

Listing 4.7: Preliminary Operators for Modular Scaling and Flooring

Provided with the non-negative integer arguments `x` and `ex`, the `shr` and `shl` operators can be interpreted to compute the right and, respectively, left bit-shift of `x` by `ex` bits. Notably, albeit these operators technically allow for `x` and `ex` to be negative, this does not occur throughout the formal verification process; as such, we can safely adopt these intuitive interpretations[2].

The remaining operators in Listing 4.7, i.e., `downscale` and `upscale`, are nearly identical to the `shr` and `shl` operators. In particular, `downscale` and `upscale` also effectively compute the right and, respectively, left bit-shift of their integer argument `x`; however, instead of a single argument that specifies the number of bits to shift, `downscale` and `upscale` both take two integer arguments `ea` and `eb` and shift for `ea - eb` bits[3]. As a last remark, although each bit-shifting operator pair in Listing 4.7 could be combined in a single operator, this would significantly complicate the remainder of the formal verification effort, especially with regards to proving the properties of these operators; therefore, we refrain from doing so.

[2]Indeed, if `x` and/or `ex` are negative, the stated bit-shift interpretations of these operators are invalid.

[3]Naturally, by extension of the fact that `shr` and `shl` are never provided a negative `x` or a negative `ex`, `downscale` and `upscale` are never given a negative `x`; moreover, `ea` will invariably be greater than or equal to `eb`.

Utilizing the preliminary operators defined above, Listing 4.8 provides two examples of modular scaling and flooring operators and their extensions: one for downwards scaling and the other for upwards scaling. As with Listing 4.6, this list of modular scaling and flooring operators is not exhaustive. That is, the actual formal verification effort comprises more of such operators than presented here; in particular, it includes variants concerning moduli different from p and q. Moreover, similarly to the modular reduction and conversion operators, all (formalizations of the) modular scaling and flooring operators follow an identical identifier format. Namely, if such an operator takes a value of type `X` and produces a value of type `Y`, the operator is identified by `scaleX2Y`.

```
(* Downwards Modular Scaling and Flooring *)
op scaleZq2Zp (z : Zq) : Zp = Zp.inzmod (downscale (Zq.asint z) eq ep).

(* Extend Downwards Modular Scaling and Flooring to Polynomials *)
op scaleRq2Rp (y : Rq) : Rp =
  BigRp.XnD1CA.bigi predT (fun (i : int) => scaleZq2Zp y.[i] ** exp Rp.iX i) 0 n.

(* Extend Downwards Modular Scaling and Flooring to Polynomial Vectors *)
op scaleRqv2Rpv (v : Rq_vec) : Rp_vec =
  Mat_Rp.Vector.offunv (fun (i : int) => scaleRq2Rp v.[i]).

(* Upwards Modular Scaling and Flooring *)
op scaleZp2Zq (z : Zp) : Zq = Zq.inzmod (upscale (Zp.asint z) eq ep).

(* Extend Upwards Modular Scaling and Flooring to Polynomials *)
op scaleRp2Rq (y : Rp) : Rq =
  BigRq.XnD1CA.bigi predT (fun i => scaleZp2Zq y.[i] ** exp Rq.iX i) 0 n.

(* Extend Upwards Modular Scaling and Flooring to Polynomial Vectors *)
op scaleRpv2Rqv (v : Rp_vec) : Rq_vec =
  Mat_Rq.Vector.offunv (fun i => scaleRp2Rq v.[i]).
```

Listing 4.8: Modular Scaling and Flooring Operators with Extensions

Respecting the definition of `downscale`, we observe that the first operator in Listing 4.8, i.e., `scaleZq2Zp`, practically performs a right bit-shift of `eq - ep` bits on the integer interpretation of its argument `z`, which is initially of type `Zq`; afterward, `scaleZq2Zp` converts this integer interpretation to `Zp`, implicitly reducing it modulo `p`. Nevertheless, this modular reduction is nullified by the preceding right bit-shift. In particular, this is because the result of bit-shifting an `eq`-bit value, such as those of type `Zq`, to the right by `eq - ep` bits cannot comprise more than `ep` bits; consequently, the succeeding reduction modulo `p` effectively reduces to the identity function. As such, `scaleZq2Zp` precisely formalizes the $\lfloor\cdot\rceil_{q\to p}$ operator defined in Section 3.1. Furthermore, the second and third operators in Listing 4.8, i.e., `scaleRq2Rp` and `scaleRqv2Rpv`, accordingly extend `scaleZq2Zp` in the same manner as the above-discussed `Rq2Rp` and `Rqv2Rpv` extend `Zq2Zp`; therefore, `scaleRq2Rp` and `scaleRqv2Rpv` formalize the coefficient-wise extension and, respectively, the combination of the coefficient-wise and entry-wise extensions of the $\lfloor\cdot\rceil_{q\to p}$ operator.

Lastly, following a similar line of reasoning to the one provided for `scaleZq2Zp`, `scaleRq2Rp`, and `scaleRqv2Rpv`, we see that the remaining three operators in Listing 4.8 formalize the $\lfloor\cdot\rceil_{p\to q}$ operator and its extensions.

Encoding and Decoding Concerning the encoding and decoding operators for the abstract cryptographic types, Listing 4.9 provides the encoding and decoding operator pairs specified for the `pkey` type as an example.

```
(* Specification Encoding and Decoding *)
op pk_encode_s : seed * Rp_vec -> pkey.
op pk_decode_s : pkey -> seed * Rp_vec.

(* Games Encoding and Decoding *)
op pk_encode_g ['a] : 'a -> pkey.
op pk_decode_g ['a] : pkey -> 'a.
```

Listing 4.9: Encoding and Decoding Operator Pairs for Public Key Type `pkey`

The first operator pair in this listing is intended for the formalization of Saber.PKE and Saber.PKEA; contrarily, the second pair is designed for the formalization of the games in the game-playing security proof. In agreement with the introduction of these encoding and decoding operators provided earlier, the operators constituting the former pair are monomorphic, i.e., they have a concrete domain and range types. Conversely, the operators composing the latter pair are polymorphic; that is, they either have an abstract domain type, an abstract range type, or both. In this case, the polymorphic encoding operator has an abstract domain type, while the polymorphic decoding operator has an abstract range type.

Exemplifying the formalization of the desired properties of the encoding and decoding operators, Listing 4.10 presents several relevant axioms; for consistency purposes, the considered axioms relate to the operators shown in Listing 4.9.

```
(* Encoding and Decoding are Each Other's Inverses *)
axiom pks_enc_dec_inv : cancel pk_encode_s pk_decode_s.
axiom pks_dec_enc_inv : cancel pk_decode_s pk_encode_s.

axiom pkg_enc_dec_inv ['a] : cancel pk_encode_g<:'a> pk_decode_g<:'a>.
axiom pkg_dec_enc_inv ['a] : cancel pk_decode_g<:'a> pk_encode_g<:'a>.


(* Encoding and Decoding Pairs are Equivalent for Correct Types *)
axiom eq_pks_pkg_enc (x : seed * Rp_vec) : pk_encode_s x = pk_encode_g x.
axiom eq_pks_pkg_dec (x : pkey) : pk_decode_s x = pk_decode_g x.
```

Listing 4.10: Properties of the Encoding and Decoding Operator Pairs for Public Key Type `pkey`

The initial four axioms in this listing denote that the encoding and decoding operators are each other's inverses. Specifically, this is achieved by means of the `cancel` predicate, which denotes that its second argument is the inverse function of its first argument. Regarding the third and fourth axiom, the `<:'a>` directly succeeding the operators ensure that the domain type of the encoding operator matches the range type of the decoding operator; particularly, this guarantees that both of these types equal the type (abstractly) represented by `'a`. Lastly, the final two axioms indicate that if the input to the polymorphic operators is of the same type as the domain of their monomorphic counterparts, then these operators are equal. Here, in `eq_pks_pkg_enc`, the concrete type of `x` automatically instantiates the domain type of `pk_encode_g` to `seed * Rp_vec`; similarly, in `eq_pks_pkg_dec`, the concrete type of `pk_decode_s x` automatically instantiates the range type of `pk_decode_g` to `seed * Rp_vec`.

Distributions

At this point, we have formalized the parameters, algebraic structures, and operators relevant to Saber.PKE. As such, the distributions employed in Saber.PKE constitute the sole artifacts still absent from the general context necessary to model Saber.PKE and its algorithms; we presently formalize these required distributions.

Comprising most of the distributions used in Saber.PKE, Listing 4.11 formalizes all distributions related to R_q.

```
(* Uniform Distribution over Rq *)
op dRq : Rq distr = Rq.dpolyXnD1.

(* Extension of Uniform Distribution over Rq to Vectors *)
op dRq_vec : Rq_vec distr = Mat_Rq.Matrix.dvector dRq.

(* Extension of Uniform Distribution over Rq to Matrices *)
op dRq_mat : Rq_mat distr = Mat_Rq.Matrix.dmatrix dRq.


(* Abstract Distribution over Rq (Replaces Centered Binomial Distribution) *)
op [lossless] dsmallRq : Rq distr.

(* Extension of Abstract Distribution over Rq to Vectors *)
op dsmallRq_vec : Rq_vec distr = Mat_Rq.Matrix.dvector dsmallRq.
```

Listing 4.11: Distributions Over R_q

In this listing, the first distribution is assigned the lossless, full, and uniform distribution over `Rq`, which is defined in the cloned instance of the polynomial quotient ring theory corresponding to this type. The second and third distributions are trivial extensions of this distribution to, respectively, vectors and matrices. Specifically, these vector and matrix distributions emerge from sampling each of their entries in accordance with the provided distribution. In this case, the provided distribution is the above-mentioned initial distribution in this listing, `dRq`. Furthermore, as is established in the `Matrix` theory, if this provided distribution is uniform, full, and lossless, then the corresponding vector and matrix distributions also possess these properties. Therefore, as desired, these vector and matrix distributions accordingly formalize the $\mathcal{U}(R_q^{l\times 1})$ and $\mathcal{U}(R_q^{l\times l})$ distributions utilized in Saber.PKE. The fourth distribution in Listing 4.11, i.e., `dsmallRq`, denotes a relatively abstract distribution over `Rq`. Namely, provided that it is lossless, `dsmallRq` may be any distribution over `Rq`. Indeed, this abstract definition of `dsmallRq` encompasses the centered binomial distribution employed in Saber.PKE. Consequently, as suggested before, proving the desired properties of Saber.PKE with this abstraction gives slightly stronger guarantees than doing so for the more concrete centered binomial distribution with parameter μ. Moreover, substituting the centered binomial distribution with this abstraction eliminates the relatively tedious effort of formalizing the centered binomial distribution. For these reasons, the formal verification effort considers `dsmallRq` instead of the centered binomial distribution with parameter μ. Lastly, analogous to `dRq_vec`, `dsmallRq_vec` is a trivial extension of `dsmallRq` to vectors. However, as opposed to `dRq_vec`, `dsmallRq_vec` only inherits the lossless property from `dsmallRq`.

Apart from the above-discussed distributions over R_q, the formal verification effort necessitates two additional distributions; these distributions correspond to $\mathcal{U}(\{0,1\}^{256})$ and $\mathcal{U}(R_p^{l\times 1})$. Listing 4.12 provides the formalizations of these distributions.

```
(* Uniform Distribution over Rp *)
op dRp : Rp distr = Rp.dpolyXnD1.

(* Extension of Uniform Distribution over Rq to Vectors *)
op dRp_vec : Rp_vec distr = Mat_Rp.Matrix.dvector dRp.

(* Uniform Distribution over Seed Domain *)
op [lossless full uniform] dseed : seed distr.
```

Listing 4.12: Distributions Over R_p and Seeds

Here, the distributions over `Rp` and `Rp_vec` are defined analogously to the distributions over `Rq` and, respectively, `Rq_vec`; as a consequence, these distributions are lossless, full, and uniform. Finally, the concrete seed domain in Saber, i.e., $\{0,1\}^{256}$, is abstractly represented by the `seed` type in the formal verification effort; as such, $\mathcal{U}(\{0,1\}^{256})$ is formalized by `dseed`, a lossless, full, and uniform distribution over `seed`.

4.2 Public-Key Encryption Scheme

Proceeding, we leverage the general context established in the preceding section to formalize and (formally) verify Saber.PKE.

This section follows an analogous structure to that of Section 3.2. That is, first, Section 4.2.1 considers the formalization of the specification provided in Section 3.2.1. Afterward, Section 4.2.2 and Section 4.2.3 respectively address the formal verification of the proofs given in Section 3.2.2 and Section 3.2.3.

4.2.1 Specification

Facilitating the formal verification process for PKE schemes, EasyCrypt's standard library provides a dedicated `PKE` theory, comprising a multitude of useful definitions. Most notably, this theory includes a module type for PKE schemes and a general IND-CPA security game based on this module type. Listing 4.13 presents the exact definition of this module type, i.e., `Scheme`.

```
module type Scheme = {
  proc kg() : pkey * skey
  proc enc(pk: pkey, m: plaintext) : ciphertext
  proc dec(sk: skey, c: ciphertext) : plaintext option
}.
```

Listing 4.13: Module Type for PKE schemes

As expected, `Scheme` defines three procedures: a key generation procedure, an encryption procedure, and a decryption procedure. Moreover, the procedures have suitable input and output types. In particular, the output type of the decryption procedure `dec` is appropriate since, for certain PKE schemes, the decryption algorithm explicitly indicates a potential "decryption failure"; this is precisely the purpose of the `option` type constructor. Specifically, this constructor can be applied to any type in order to construct a slightly extended type that, in addition to all the values of the original type, includes the value `None`. Indeed, this `None` is analogous to the Null value in many programming languages, enabling, for example, the explicit indication of a failure. As a minor syntactical change that comes with the application of this type constructor, values of the constructed type that correspond to values of the original type are referred to with a prepended "`Some`". For instance, if `m` is a value of type `plaintext`, then `Some m` is the corresponding value of type `plaintext option`. Nevertheless, although utilized in many cryptographic constructions, the explicit indication of failures is not described in Saber.Dec's specification; hence, the return value of (the formalization of) Saber.PKE's decryption algorithm is invariably of the form `Some x`, where `x` is a value of type `plaintext`.

Utilizing the `Scheme` module type, Listing 4.14 gives the skeleton structure of Saber.PKE's formalization.

```
module Saber_PKE_Scheme : Scheme = {
   proc kg() : pkey * skey = {

   }

   proc enc(pk: pkey, m: plaintext) : ciphertext = {

   }

   proc dec(sk: skey, c: ciphertext) : plaintext option = {

   }
}.
```

Listing 4.14: Skeleton Structure of Saber.PKE

Next, Listing 4.15 adds the concrete implementation of the `kg` procedure to this skeleton structure. This implementation is obtained by, after declaring the necessary variables, directly translating the specification of Saber.KeyGen provided in Section 3.2.1. Here, the `*^` operator denotes matrix-vector multiplication.

```
module Saber_PKE_Scheme : Scheme = {
   proc kg() : pkey * skey = {
      var sd: seed;
      var _A: Rq_mat;
      var s: Rq_vec;
      var b: Rp_vec;

      sd <$ dseed;
      _A <- gen sd;
      s <$ dsmallRq_vec;
      b <- scaleRqv2Rpv (_A *^ s + h);

      return (pk_encode_s (sd, b), sk_encode_s s);
   }

   proc enc(pk: pkey, m: plaintext) : ciphertext = {

   }

   proc dec(sk: skey, c: ciphertext) : plaintext option = {

   }
}.
```

Listing 4.15: Saber.KeyGen's Specification

In order to comply with the `Scheme` module type, rather than immediately returning the concrete values computed for the public and secret keys, we first convert these values to the associated abstract cryptographic types, i.e., `pkey` and `skey`; indeed, this is accomplished by means of the corresponding encoding operators.

Extending the formalization of Saber.PKE, Listing 4.16 includes the implementation of the `enc` procedure. However, to prevent unnecessary duplication, this listing folds the implementation of the `kg` procedure, as indicated by the `(*...*)`. The implementation of `enc` utilizes several operators not elaborated on before, viz. `trmx` and `dotp`. These operators compute the transpose of its matrix argument and the inner product of its two vector arguments, respectively.

```
module Saber_PKE_Scheme : Scheme = {
   proc kg() : pkey * skey = {
      (*...*)
   }

   proc enc(pk: pkey, m: plaintext) : ciphertext = {
      var pk_dec: seed * Rp_vec;
      var m_dec: R2;
      var sd: seed;
      var _A: Rq_mat;
      var s': Rq_vec;
      var b, b': Rp_vec;
      var v': Rp;
      var cm: R2t;

      m_dec <- m_decode m;
      pk_dec <- pk_decode_s pk;
      sd <- pk_dec.`1;
      b <- pk_dec.`2;

      _A <- gen sd;
      s' <$ dsmallRq_vec;
      b' <- scaleRqv2Rpv ((trmx _A) *^ s' + h);
      v' <- (dotp b (Rqv2Rpv s')) + (Rq2Rp h1);
      cm <- scaleRp2R2t (v' + (scaleR22Rp m_dec));

      return c_encode_s (cm, b');
   }

   proc dec(sk: skey, c: ciphertext) : plaintext option = {

   }
}.
```

Listing 4.16: Saber.Enc's Specification

Naturally, because the input arguments of `enc` are of abstract cryptographic types, they must be decoded to concrete values before being usable in the procedure's computations. Likewise, as with the `kg` procedure, the return values of `enc` must be encoded to their respective abstract cryptographic types in order to comply with the `Scheme` module type.

Finally, completing Saber.PKE's formalization, Listing 4.17 incorporates the implementation of the `dec` procedure. To preclude redundancy, this listing folds the implementations of the `kg` and `enc` procedures.

```
module Saber_PKE_Scheme : Scheme = {
   proc kg() : pkey * skey = {
      (*...*)
   }

   proc enc(pk: pkey, m: plaintext) : ciphertext = {
      (*...*)
   }

   proc dec(sk: skey, c: ciphertext) : plaintext option = {
      var c_dec: R2t * Rp_vec;
      var cm: R2t;
      var b': Rp_vec;
      var v: Rp;
      var s: Rq_vec;
      var m': R2;

      c_dec <- c_decode_s c;
      s <- sk_decode_s sk;
      cm <- c_dec.`1;
      b' <- c_dec.`2;

      v <- (dotp b' (Rqv2Rpv s)) + (Rq2Rp h1);
      m' <- scaleRp2R2 (v  - (scaleR2t2Rp cm) + (Rq2Rp h2));

      return Some (m_encode m');
    }
};
```

Listing 4.17: Saber.Dec's Specification

As elaborated on earlier, due to the `option` type constructor, the return value of the `dec` procedure must be prepended by `Some`. Apart from this, the implementation of `dec` does not contain any unprecedented concepts in need of further elaboration.

4.2.2 Security

Thus far, we have formalized the relevant general context and the specification of Saber.PKE. Employing these formalizations, we can initiate the actual formal verification process for Saber.PKE's desired properties; as in Chapter 3, we commence with Saber.PKE's IND-CPA security property. To this end, the ensuing discussion follows a similar structure to that of Section 3.2.2. Namely, we first address the formalization of the relevant hardness assumptions and security property. Subsequently, we cover the formalization and formal verification of the proofs related to these hardness assumptions and this security property.

Security Property and Hardness Assumptions

As aforementioned, the `PKE` theory provided in EasyCrypt's standard library defines a general IND-CPA security game; in essence, this is a formalization of Game$^{\text{IND-CPA}}_{\mathcal{A},\text{PKE}}$, depicted in Figure 3.1. Listing 4.18 comprises the precise definition of this (formalized) game.

```
module CPA (S : Scheme, A : Adversary) = {
  proc main() : bool = {
    var pk : pkey;
    var sk : skey;
    var m0, m1 : plaintext;
    var c : ciphertext;
    var b, b' : bool;

    (pk, sk) <@ S.kg();
    (m0, m1) <@ A.choose(pk);
    (* {0,1} denotes a lossless, full, and uniform distribution over the boolean
    values false and true *)
    b        <$ {0,1};
    c        <@ S.enc(pk, if (b) then m1 else m0);
    b'       <@ A.guess(c);

    return (b' = b);
  }
}.
```

Listing 4.18: Game$^{\text{IND-CPA}}_{\mathcal{A},\text{PKE}}$

As can be derived from this listing, the general IND-CPA game is defined with respect to two module arguments: the first is of module type `Scheme` and the second is of module type `Adversary`. Indeed, the former module type is the one utilized for Saber.PKE's formalization; the latter module type is novel and, for this reason, specified in Listing 4.19.

```
module type Adversary = {
  proc choose(pk : pkey) : plaintext * plaintext
  proc guess(c : ciphertext) : bool
}.
```

Listing 4.19: Module Type for Adversaries Against $\text{Game}^{\text{IND-CPA}}_{\mathcal{A},\text{PKE}}$

Evidently, the `Adversary` module type formalizes the class of adversaries against the IND-CPA game. More precisely, the `choose` and `guess` procedures respectively model the $\mathcal{A}.\mathsf{P}$ and $\mathcal{A}.\mathsf{D}$ algorithms defined by adversaries from this class. Notably however, despite their correspondence, $\mathcal{A}.\mathsf{D}$ and `guess` differ with respect to their parameters. Namely, `guess` only takes a ciphertext, while $\mathcal{A}.\mathsf{D}$ takes both a public key and a ciphertext. Nevertheless, because the formal verification effort exclusively considers these adversaries in the context of the above-introduced `CPA` module, an adversary's `choose` procedure is invariably called before its `guess` procedure. Furthermore, no computational restrictions are imposed on the considered adversaries; in particular, these adversaries are capable of maintaining states. As such, an adversary can store the value of the public key `pk` provided in the call to its `choose` procedure, removing the need of passing this value anew in the subsequent call to its `guess` procedure[4].

From the above, we can derive that the `CPA` module may be instantiated with `Saber_PKE_Scheme` and any module of type `Adversary`. Moreover, doing so produces a correct formalization of the concrete IND-CPA game for Saber.PKE and an IND-CPA adversary $\mathcal{A} = (\mathsf{P}, \mathsf{D})$, i.e., $\text{Game}^{\text{IND-CPA}}_{\mathcal{A},\text{Saber.PKE}}$.

Regarding the utilized hardness assumptions, we initiate the formalization process by considering the various relevant adversaries. Examining the definitions of these hardness assumptions in Figure 3.4 and Figure 3.5, we see that adversaries against the GMLWR game have different interfaces from adversaries against the XMLWR game; as such, we specify a distinct module type for each of these adversary classes. Listing 4.20 provides these module types.

```
module type Adv_GMLWR = {
   proc guess(sd : seed, b : Rp_vec) : bool
}.

module type Adv_XMLWR = {
   proc guess(sd : seed, b : Rp_vec, a : Rq_vec, d : Rp) : bool
}.
```

Listing 4.20: Module Types for Adversaries Against $\text{Game}^{\text{GMLWR}}_{\mathcal{A},l,\mu,q,p}$ and $\text{Game}^{\text{XMLWR}}_{\mathcal{A},l,\mu,q,p}$

As suggested by their names and interfaces, from top to bottom, the module types in this listing respectively represent (the classes of) adversaries against $\text{Game}^{\text{GMLWR}}_{\mathcal{A},l,\mu,q,p}$ and $\text{Game}^{\text{XMLWR}}_{\mathcal{A},l,\mu,q,p}$.

Finally, due to the similarity between the GMLWR and XMLWR games, we solely present the formalization of the GMLWR game, i.e., $\text{Game}^{\text{GMLWR}}_{\mathcal{A},l,\mu,q,p}$. This formalization, `GMLWR`, is particularized in Listing 4.21. Given the concepts introduced and explained in the preceding discussion, `GMLWR` is a rather straightforward translation of $\text{Game}^{\text{GMLWR}}_{\mathcal{A},l,\mu,q,p}$, only significantly deviating from a verbatim translation in a single aspect. Namely, rather than a formalization of the modular scaling and

[4]Certainly, in $\text{Game}^{\text{IND-CPA}}_{\mathcal{A},\text{PKE}}$, the public key pk is technically not required as an argument to $\mathcal{A}.\mathsf{D}$ as well; however, for reasons of explicitness, $\text{Game}^{\text{IND-CPA}}_{\mathcal{A},\text{PKE}}$ still passes the public key in the call to this algorithm.

rounding operator employed in $\text{Game}^{\text{GMLWR}}_{\mathcal{A},l,\mu,q,p}$, `GMLWR` utilizes the formalization of the analogous modular scaling and flooring operator under the addition of `h`; here, `h` formalizes the constant $\mathbf{h}$. Furthermore, we do not explicitly verify that (the formalizations of) these operations are equivalent; that is, we essentially assume that for all $\mathbf{x} \in R_q^{l\times 1}$, $\lfloor \mathbf{x} \rceil_{q\to p}$ is indeed equivalent to $\lfloor \mathbf{x} + \mathbf{h} \rfloor_{q\to p}$, as shown in Section 3.1. The rationale for doing so arises from the severe disparity between the triviality of this equivalence and the effort necessary to formally verify it. Thus, we directly formalize both the GMLWR and XMLWR games with the modular scaling and flooring operator.

```
module GMLWR(A : Adv_GMLWR) = {
   proc main(u : bool) : bool = {
      var u' : bool;
      var sd : seed;
      var _A : Rq_mat;
      var s : Rq_vec;
      var b : Rp_vec;

      sd <$ dseed;
      _A <- gen sd;
      s <$ dsmallRq_vec;

      if (u) {
         b <$ dRp_vec;
      } else {
         b <- scaleRqv2Rpv (_A *^ s + h);
      }

      u' <@ A.guess(sd, b);

      return u';
   }
}.
```

Listing 4.21: $\text{Game}^{\text{GMLWR}}_{\mathcal{A},l,\mu,q,p}$

Hardness of GMLWR and XMLWR

Before using the above-discussed formalizations of the GMLWR and XMLWR games, i.e., `GMLWR` and `XMLWR`, in the formal verification of Saber.PKE's IND-CPA security property, we argue for the aptness of these hardness assumptions. In particular, we do so by formally verifying the corresponding ROM proofs discussed in Section 3.2.2; that is, we formally verify that if `gen` is a random oracle, (instances of) GMLWR and XMLWR are at least as hard as (corresponding instances of) MLWR.

Due to the ubiquity of the ROM in cryptography, EasyCrypt supplies a multitude of theories related to this concept. One of these theories provides the definitions and properties associated with the concept of a *Programmable Random Oracle (PRO)*. Essentially, such a random oracle allows for the manual manipulation of the mapping it defines. Indeed, this suffices to model the manipulation of the random oracle query results performed by the reduction adversaries in the ROM proofs concerning the hardness of the GMLWR and XMLWR games. Listing 4.22 provides the relevant part of the module type utilized to formalize these types of random oracles in EasyCrypt.

```
module type PRO = {
  proc init()
  proc get(x : in_t) : out_t
  proc set(x : in_t, y : out_t)
  (*...*)
}.
```

Listing 4.22: Module Type for Programmable Random Oracles

Here, `in_t` and `out_t` types are abstract placeholders for the oracle's input and output types, respectively. In the current context, since the employed oracle replaces the `gen` function, `in_t` is instantiated with `seed` and `out_t` is instantiated with `Rq_mat`. Furthermore, the intended purpose of each provided procedure is quite evident from its signature. Specifically, `init()` performs the necessary initialization of the oracle, `get(x)` returns the image of `x`, and `set(x, y)` sets the image of `x` to `y`. As such, `get` embodies the querying of the random oracle, while `set` represents the manipulation of the oracle query results.

Demonstrating the formal verification of the hardness proofs, we consider the proof regarding GMLWR. As discussed in Section 3.2.2, given an adversary $\mathcal{A}$ against $\text{GameROM}^{\text{GMLWR}}_{\mathcal{A},l,\mu,q,p}$, this proof constructs an adversary $\mathcal{R}^{\mathcal{A}}$ against $\text{Game}^{\text{MLWR}}_{\mathcal{R}^{\mathcal{A}},l,l,\mu,q,p}$. Hence, for the formal verification of this proof, we foremost require a formalization of these (classes of) adversaries and games. Akin to before, the classes of adversaries are formalized through dedicated module types; Listing 4.23 defines these module types.

```
module type Adv_MLWR = {
  proc guess(_A : Rq_mat, b : Rp_vec) : bool
}.

module type Adv_GMLWR_RO(Gen : PRO) = {
   proc guess(sd : seed, b : Rp_vec) : bool { Gen.get }
}.
```

Listing 4.23: Module Types for Adversaries Against $\text{Game}^{\text{MLWR}}_{\mathcal{A},l,l,\mu,q,p}$ and $\text{GameROM}^{\text{GMLWR}}_{\mathcal{A},l,\mu,q,p}$

From top to bottom, these module types respectively represent the classes of adversaries against $\text{Game}^{\text{MLWR}}_{\mathcal{A},l,l,\mu,q,p}$ and $\text{GameROM}^{\text{GMLWR}}_{\mathcal{A},l,\mu,q,p}$. Concerning the latter, the explicit random oracle access is modeled through a module parameter of the above-mentioned `PRO` type. Nevertheless, since these adversaries are solely allowed to query the random oracle, they exclusively gain access to the random oracle's `get` procedure. Indeed, this is conveyed to EasyCrypt by the `{ Gen.get }` in the definition of the corresponding `guess` procedure.

As with the preceding game formalizations, we formalize $\text{Game}^{\text{MLWR}}_{\mathcal{A},l,l,\mu,q,p}$ and $\text{GameROM}^{\text{GMLWR}}_{\mathcal{A},l,\mu,q,p}$ through specifying a separate module for each of them; both of these modules are parameterized by other modules that formalize the adversaries against the corresponding games. More precisely, the formalization of $\text{Game}^{\text{MLWR}}_{\mathcal{A},l,l,\mu,q,p}$, `MLWR`, is defined with respect to a parameter of type `Adv_MLWR`; analogously, the formalization of $\text{GameROM}^{\text{GMLWR}}_{\mathcal{A},l,\mu,q,p}$, `GMLWR_RO`, is defined with respect to a parameter of type `Adv_GMLWR_RO`. Apart from its parameter type, this latter formalization solely deviates from `GMLWR` in the acquisition of `_A`; specifically, while `GMLWR` evaluates `gen sd`, `GMLWR_RO` queries the random oracle on `sd`. Certainly, this is consistent with the differences between $\text{GameROM}^{\text{GMLWR}}_{\mathcal{A},l,\mu,q,p}$ and $\text{Game}^{\text{GMLWR}}_{\mathcal{A},l,\mu,q,p}$. Due to its vast similarity with `GMLWR`, `GMLWR_RO` is not explicitly presented here. Contrarily, albeit a rather trivial translation of $\text{Game}^{\text{MLWR}}_{\mathcal{A},l,l,\mu,q,p}$, `MLWR` is specified in Listing 4.24 for future reference and reasons of completeness.

```
module MLWR(A : Adv_MLWR) = {
    proc main(u : bool) : bool = {
       var u' : bool;
       var _A : Rq_mat;
```

```
      var s : Rq_vec;
      var b : Rp_vec;

      _A <$ dRq_mat;
      s <$ dsmallRq_vec;

      if (u) {
         b <$ dRp_vec;
      } else {
         b <- scaleRqv2Rpv (_A *^ s + h);
      }

      u' <@ A.guess(_A, b);

      return u';
   }
}.
```

Listing 4.24: $\text{Game}^{\text{MLWR}}_{\mathcal{A},l,l,\mu,q,p}$

Then, utilizing the above-introduced module types, Listing 4.25 presents the formalization of the reduction adversary $\mathcal{R}^{\mathcal{A}}$ against $\text{Game}^{\text{MLWR}}_{\mathcal{R}^{\mathcal{A}},l,l,\mu,q,p}$, where $\mathcal{A}$ is any adversary against $\text{GameROM}^{\text{GMLWR}}_{\mathcal{A},l,\mu,q,p}$.

```
(* Adversary Against MLWR (l samples) Game, Constructed From Adversary Against
    GMLWR_RO Game *)
module AGM(AG : Adv_GMLWR_RO) : Adv_MLWR = {
   module AG = AG(Gen)

   proc guess(_A : Rq_mat, b : Rp_vec) : bool = {
      var u' : bool;
      var sd : seed;

      Gen.init();

      sd <$ dseed;

      Gen.set(sd, _A);

      u' <@ AG.guess(sd, b);

      return u';
   }
}.
```

Listing 4.25: Reduction Adversary $\mathcal{R}^{\mathcal{A}}$ Against $\text{Game}^{\text{MLWR}}_{\mathcal{R}^{\mathcal{A}},l,l,\mu,q,p}$

In this listing, `Gen` constitutes a concrete implementation of the required random oracle[5]; more precisely, `Gen` is a module of type `PRO`, where the `in_t` and `out_t` types are accordingly instantiated with the `seed` and `Rq_mat` types. Using this random oracle, we instantiate the given adversary through the statement `module AG = AG(Gen)`. Indeed, this instantiation provides adversary `AG` access to `Gen`; nevertheless, due to the restriction specified in the `Adv_GMLWR_RO` module type, `AG` is only capable of accessing the `get` procedure of `Gen`. Employing this instantiated adversary against `GMLWR_RO`, (the formalization of) the reduction adversary, i.e., `AGM`, operates as follows. Foremost, `AGM` initializes the random oracle by means of a call to `Gen.init()`; intuitively, this can be interpreted as the random oracle defining a uniformly random mapping from its input to its output

[5]This implementation is provided in EasyCrypt's standard library.

domain. Afterward, `AGM` samples a value of type `seed` uniformly at random, storing the result in `sd`. Subsequently, `AGM` adjusts the mapping defined by the random oracle; specifically, it ensures that the random oracle maps `sd` to `_A`, i.e., the matrix received from the `MLWR` game. Penultimately, `AGM` calls `AG.guess(sd, b)` in an effort to solve the GMLWR problem instance corresponding to `sd` and `b`. Indeed, since the random oracle maps `sd` to `_A`, this GMLWR problem instance exactly matches the MLWR problem instance that `AGM` attempts to solve. Moreover, because the random oracle's output distribution remains uniformly random, `AG` cannot distinguish between the reduction and the corresponding run of its own game; in turn, as elaborated on before, this guarantees that `AG` behaves as in this corresponding run of `GMLWR_RO`. Concluding, directly returning the value obtained from the call to `AG.guess(sd, b)`, as `AGM` ultimately does, results in a *winning probability* for `AGM` against the `MLWR` instance with `_A` and `b` that is equal to the *winning probability* of `AG` against the `GMLWR_RO` instance with `sd` and `b`. In fact, since this reasoning is independent of the value of `u` (from `MLWR`), this process additionally guarantees an *advantage* for `AGM` that is equal to the *advantage* of `AG`.

Lastly, the formal verification effort concerning the equality between the advantages of $\mathcal{A}$ against $\text{GameROM}^{\text{GMLWR}}_{\mathcal{A},l,\mu,q,p}$ and $\mathcal{R}^{\mathcal{A}}$ against $\text{Game}^{\text{MLWR}}_{\mathcal{R}^{\mathcal{A}},l,l,\mu,q,p}$ commences by formalizing this property through an appropriately defined lemma. Listing 4.26 defines an instance of such a lemma.

```
lemma Equal_Advantage_GMLWR_RO_MLWR &m (A <: Adv_GMLWR_RO{Gen}) :
  `| Pr[GMLWR_RO(A).main(true) @ &m : res] - Pr[GMLWR_RO(A).main(false) @ &m : res] |
  =
  `| Pr[MLWR( AGM(A) ).main(true) @ &m : res] - Pr[MLWR( AGM(A) ).main(false) @ &m :
    res] |.
proof. (*...*) qed.
```

Listing 4.26: Equality of Advantages for $\mathcal{A}$ Against $\text{GameROM}^{\text{GMLWR}}_{\mathcal{A},l,\mu,q,p}$ and $\mathcal{R}^{\mathcal{A}}$ Against $\text{Game}^{\text{MLWR}}_{\mathcal{R}^{\mathcal{A}},l,l,\mu,q,p}$

For a detailed deconstruction and explanation of lemmas such as `Equal_Advantage_GMLWR_RO_MLWR`, refer to the subsequent discussion on the formal verification of the game-playing security proof. At present, it suffices to recognize the following correspondences.

`Pr[GMLWR_RO(A).main(true) @ &m : res]`	$\cong$	$\Pr\left[\text{GameROM}^{\text{GMLWR}}_{\mathcal{A},l,\mu,q,p}(1) = 1\right]$
`Pr[GMLWR_RO(A).main(false) @ &m : res]`	$\cong$	$\Pr\left[\text{GameROM}^{\text{GMLWR}}_{\mathcal{A},l,\mu,q,p}(0) = 1\right]$
`Pr[MLWR( AGM(A) ).main(true) @ &m : res]`	$\cong$	$\Pr\left[\text{Game}^{\text{MLWR}}_{\mathcal{R}^{\mathcal{A}},l,l,\mu,q,p}(1) = 1\right]$
`Pr[MLWR( AGM(A) ).main(false) @ &m : res]`	$\cong$	$\Pr\left[\text{Game}^{\text{MLWR}}_{\mathcal{R}^{\mathcal{A}},l,l,\mu,q,p}(0) = 1\right]$

From the `(A <: Adv_GMLWR_RO{Gen})` before the colon of `Equal_Advantage_GMLWR_RO_MLWR`, we discern that in the above probability statements, `A` is an adversary against `GMLWR_RO`; moreover, as enforced by the `{Gen}`, `A` is unable to access the global variables of the random oracle, which include the random oracle's concrete mapping. Then, considering `` `| | `` denotes the absolute value operator, we see that the statement succeeding the colon of `Equal_Advantage_GMLWR_RO_MLWR` correctly formalizes the equality between the considered advantages.

The proof of lemma `Equal_Advantage_GMLWR_RO_MLWR`, which was left out in Listing 4.26, proceeds in a straightforward manner. Namely, the truth of this lemma trivially follows after showing that `Pr[GMLWR_RO(A).main(true) @ &m : res]` equals `Pr[MLWR( AGM(A) ).main(true) @ &m : res]` and `Pr[GMLWR_RO(A).main(false) @ &m : res]` equals `Pr[MLWR( AGM(A) ).main(false) @ &m : res]`. In turn, these equalities follow from the fact that the collated games are indistinguishable from the perspective of `A`. Certainly, as aforementioned, this is a consequence of the uniformity of the random oracle query results observed by `A` (i.e., even after the manipulation of these query results performed by the reduction adversary).

IND-CPA **Security of** Saber.PKE

Hitherto, by leveraging the initially established general context, we have formalized the specification of Saber.PKE, the IND-CPA game for Saber.PKE, the GMLWR and XMLWR games, and the relevant classes of adversaries against these games. With respect to Saber.PKE's IND-CPA security, these constitute the required concepts not specific to the corresponding game-playing security proof. As such, proceeding, we first formalize the necessary concepts specific to this proof. More precisely, we extend the general context to include the assumption of the security theorem, i.e., $\frac{q}{p} \leq \frac{p}{2 \cdot t}$; furthermore, we formalize the sequence of games and reduction adversaries utilized in the security proof. Afterward, employing the constructed formalizations, we formalize the steps carried out in the game-playing security proof. Ultimately, we formally verify the correctness of the proof and, hence, the validity of the security theorem.

Proof-Specific Context The security theorem in Section 3.2.2 extends the general context of Saber.PKE by introducing an additional assumption; indeed, this assumption states that $\frac{q}{p} \leq \frac{p}{2 \cdot t}$. The context emerging from this extension constitutes the full context in which the security proof of Saber.PKE manifests itself, i.e., the proof-specific context. To accurately replicate this context in EasyCrypt, we must formalize this assumption as well; this formalization is provided in Listing 4.27. Notice that, since $p \mid q$ and $2 \cdot t \mid p$, evaluating $\frac{q}{p}$ and $\frac{p}{2 \cdot t}$ through regular division is equivalent to computing their respective quotients arising from Euclidean division. As such, $\frac{q}{p} \leq \frac{p}{2 \cdot t}$ can be formalized using the `%/` operator.

```
axiom sec_assumption_og: q %/ p <= p %/ (2 * t).
```

Listing 4.27: Security Theorem Assumption

Game Sequence Having established the proof-specific context, we continue by formalizing the game sequence corresponding to the considered game-playing security proof, i.e., the game sequence presented in Figure 3.8; specifically, we do so by defining a separate module for each game in the sequence. All of these modules are defined with respect to a module parameter of type `Adversary`, modeling the considered IND-CPA adversary $\mathcal{A} = (\mathsf{P}, \mathsf{D})$. Given all of the previously discussed material, the remainder of each module definition is a relatively straightforward translation of the corresponding game's specification; as such, the complete formalizations of these games are not explicitly presented here. However, for future reference, we remark that in each of these formalizations, the procedure comprising the actual implementation of the corresponding game has the signature `main() : bool`. Furthermore, throughout the remainder (with the exception of code listings), the formalizations of $\mathrm{Game}^0_{\mathcal{A}}$, $\mathrm{Game}^1_{\mathcal{A}}$, $\mathrm{Game}^2_{\mathcal{A}}$, $\mathrm{Game}^3_{\mathcal{A}}$, and $\mathrm{Game}^4_{\mathcal{A}}$ are respectively referred to as `Game0`, `Game1`, `Game2`, `Game3`, and `Game4`.

Reduction Adversaries Akin to the games in the game sequence, the considered reduction adversaries are formalized as distinct modules. As suggested in Section 3.2.2, these reduction adversaries can essentially be divided into two mutually exclusive categories. Namely, one category concerns reductions between two consecutive games, the other category regards reductions from distinguishing between two successive games to a hardness assumption. Exemplifying the formalization of reduction adversaries from the former category, Listing 4.28 defines the module that formalizes reduction adversary $\mathcal{R}^{\mathcal{A}}$ against $\mathrm{Game}^2_{\mathcal{R}^{\mathcal{A}}}$, specified in Figure 3.10.

```
(* Adversary A2 Against Game2, Constructed from Adversary A1 Against Game1 *)
module A2(A1 : Adversary) : Adversary = {
  proc choose(pk : pkey) : plaintext * plaintext = {
    var m0, m1 : plaintext;
```

```

      (m0, m1) <@ A1.choose(pk);

      return (m0, m1);
   }

   proc guess(c : ciphertext) : bool = {
      var u' : bool;
      var c_dec : Rppq * Rp_vec;
      var cmu : Rppq;
      var b' : Rp_vec;
      var cmu' : R2t;

      c_dec <- c_decode_g c;
      cmu <- c_dec.`1;
      b' <- c_dec.`2;

      cmu' <- scaleRppq2R2t cmu;

      u' <@ A1.guess(c_encode_g (cmu', b'));

      return u';
   }
}.
```

Listing 4.28: Reduction Adversary $\mathcal{R}^{\mathcal{A}}$ Against $\text{Game}^2_{\mathcal{R}^{\mathcal{A}}}$

As Listing 4.28 conveys, reduction adversary $\mathcal{R}^{\mathcal{A}}$ against $\text{Game}^2_{\mathcal{R}^{\mathcal{A}}}$ is formalized as module `A2`, parameterized by a different module `A1`. Moreover, both `A1` and `A2` are of type `Adversary`, ensuring they are suitable as arguments to their intended games. As desired, the `A2.choose` and `A2.guess` procedures are, respectively, exact translations of the $\mathcal{A}'.\mathsf{P}$ and $\mathcal{A}'.\mathsf{D}$ algorithms presented in Figure 3.10, i.e., barring the necessary encoding and decoding operations concerning the abstract cryptographic types.

Illustrating the formalization of reduction adversaries from the other category, Listing 4.29 provides the formalization of $\mathcal{B}_0^{\mathcal{A}}$ against $\text{Game}^{\text{GMLWR}}_{\mathcal{B}_0^{\mathcal{A}},l,\mu,q,p}$, defined in Figure 3.9.

```
(* Adversary B0 Against GMLWR, Constructed from Adversary A Distinguishing Between
    Game0 and Game1 *)
module B0(A : Adversary) : Adv_GMLWR = {
   proc guess(sd : seed, b : Rp_vec) : bool = {
      var w, w' : bool;
      var m0, m1 : plaintext;
      var _A : Rq_mat;
      var s' : Rq_vec;
      var b' : Rp_vec;
      var v' : Rp;
      var cmw : R2t;

      w <$ dbool;
      _A <- gen sd;

      (m0, m1) <@ A.choose(pk_encode_g (sd, b));

      s' <$ dsmallRq_vec;
      b' <- scaleRqv2Rpv ((trmx _A) *^ s' + h);
      v' <- (dotp b (Rqv2Rpv s')) + (Rq2Rp h1);
```

```

        if (w) {
           cmw <- scaleRp2R2t (v' + (scaleR22Rp (m_decode m1)));
        } else {
           cmw <- scaleRp2R2t (v' + (scaleR22Rp (m_decode m0)));
        }

        w' <@ A.guess(c_encode_g (cmw, b'));

        return (w = w');
    }
}.
```

Listing 4.29: Reduction Adversary $\mathcal{B}_0^{\mathcal{A}}$ Against $\text{Game}^{\text{GMLWR}}_{\mathcal{B}_0^{\mathcal{A}},l,\mu,q,p}$

Similarly to reduction adversary $\mathcal{R}^{\mathcal{A}}$, $\mathcal{B}_0^{\mathcal{A}}$ is formalized as a module `B0` parameterized by another module `A`. However, in contrast to `A1`, `B0` is of type `Adv_GMLWR`; naturally, this correctly corresponds to the fact that $\mathcal{B}_0^{\mathcal{A}}$ is an adversary against $\text{Game}^{\text{GMLWR}}_{\mathcal{B}_0^{\mathcal{A}},l,\mu,q,p}$. Likewise, the `Adversary` type of `A` justly models the fact that the $\mathcal{A}$ in $\mathcal{B}_0^{\mathcal{A}}$ is an IND-CPA adversary. Lastly, concerning the concrete implementation of `B0`, the `guess` procedure is virtually a direct translation of the algorithm of $\mathcal{B}_0^{\mathcal{A}}$ specified in Figure 3.9.

Auxiliary Games At this point, we have, in addition to the entire proof-specific context, formalized every concept employed in Section 3.2.2; nevertheless, before advancing to the formal verification of the game-playing security proof, we define several auxiliary games. The reason for this is that, while unnecessarily verbose for the discussion in Section 3.2.2, these auxiliary games significantly reduce the complexity of the security proof's formal verification. For each of these auxiliary games, we describe its underlying mechanisms and specific purpose in the formal verification of the security proof. However, we do not provide listings with the precise definitions of these games in EasyCrypt, predominantly because this would hardly convey any additional valuable information for the current discussion[6].

The initial two auxiliary games, `Game2a` and `Game2b`, facilitate in verifying the correctness of the step from `Game2` to `Game3`; in particular, they do so by enabling the use of EasyCrypt's `DMapSampling` theory. Conceptually, this theory proves that, given a distribution $\mathcal{D}(X)$ and a function $f : X \to Y$, sampling from $\mathcal{D}$ and then applying f is equivalent to directly sampling from the distribution that results from mapping $\mathcal{D}(X)$ with respect to f. Indeed, for $\mathcal{D}(X) := \mathcal{U}(R_q^{l\times 1})$ and $f := \bmod p$, this is exactly one of the properties required to prove the validity of the step from $\text{Game}_{\mathcal{A}}^2$ to $\text{Game}_{\mathcal{A}}^3$. Specifically, as shown in Section 3.2.2, the step from $\text{Game}_{\mathcal{A}}^2$ to $\text{Game}_{\mathcal{A}}^3$ is correct (partly) due to the fact that sampling from $\mathcal{U}(R_q^{l\times 1})$ and then reducing modulo p is equivalent to directly sampling from $\mathcal{U}(R_p^{l\times 1})$. Certainly, $\mathcal{U}(R_p^{l\times 1})$ is the distribution that results from mapping $\mathcal{U}(R_q^{l\times 1})$ with respect to reduction modulo p.

The final auxiliary game, `Auxiliary_Game`, assists in proving that `Game4` guarantees a winning probability of exactly $\frac{1}{2}$, independent of the considered adversary. These games, i.e., `Auxiliary_Game` and `Game4`, are nearly identical; in fact, they solely deviate in how they obtain the ciphertext and the order of certain operations. Specifically, while `Game4` computes the first part of its ciphertext through an addition involving an adversarially selected message, `Auxiliary_Game` samples this artifact uniformly at random. Moreover, rather than in its initial statement, `Auxiliary_Game` samples the selection bit in the statement directly preceding its final return statement[7]. This is possible because, apart from the final return statement, no statement requires knowledge of this bit. As such, it is trivial to verify that all computations in `Auxiliary_Game` are independent of the selection bit.

[6]If so desired, refer to the actual code for the concrete definitions of these auxiliary games in EasyCrypt.

[7]In terms of the games from the security proof's game sequence (see Figure 3.8), "selection bit" refers to bit u.

In particular, this implies that all artifacts passed to the adversary provide no information about the selection bit; hence, an adversary against `Auxiliary_Game` invariably has a winning probability of $\frac{1}{2}$. Although it is relatively complicated to formally verify that, in `Game4`, all artifacts provided to the adversary are completely independent of the selection bit, it is significantly less difficult to formally verify the equivalence between `Game4` and `Auxiliary_Game`. From this equivalence, it follows that the winning probability for any adversary against `Game4` equals $\frac{1}{2}$ as well.

Security Proof Leveraging the established formalizations, the remainder of this discussion covers the formal verification of the game-playing security proof. To this end, we illustrate the formal verification process for each kind of proof step with a concrete example from the actual formal verification effort; naturally, as can be extracted from Section 3.2.2, these kinds of proof step directly correspond to the aforementioned categories of reduction adversaries. For intelligibility purposes, the presented examples are consistent with the instances of reduction adversaries described earlier; that is, the considered proof steps utilize the reduction adversaries from Listing 4.28 and Listing 4.29. Although not all four proof steps are explicated individually, the discussion on a particular step can straightforwardly be extrapolated to other steps of the same kind. After the consideration of these proof steps, we detail the formal verification of the proof's remnants; specifically, these remnants comprise the equivalence between $\text{Game}^{\text{IND-CPA}}_{\mathcal{A},\text{Saber.PKE}}$ and $\text{Game}^{0}_{\mathcal{A}}$, the invariable winning probability of any adversary against $\text{Game}^{4}_{\mathcal{A}}$, and the security theorem.

Demonstrating the formalization of the different kinds of proof step, Listing 4.30 provides the lemmas stating the desired results of the security proof's first and second step. As indicated by the `&m` and `(A <: Adversary)` preceding the colons, both lemmas in this listing universally quantify over the sets of potential memories and relevant adversaries[8]. Regarding the latter, this type is explicitly indicated to be `Adversary`, in agreement with the module parameters of the considered games and reduction adversaries. Consequently, for each lemma to hold, the statement after its colon must be true for all possible combinations of a valid memory and a module of type `Adversary`.

```
(* Step 1 *)
lemma Step_Distinguish_Game0_Game1_GMLWR &m  (A <: Adversary) :
  `| Pr[Game0(A).main() @ &m : res] - Pr[Game1(A).main() @ &m : res] |
  =
  `| Pr[GMLWR( B0(A) ).main(true) @ &m : res] - Pr[GMLWR( B0(A) ).main(false) @ &m :
    res] |.
proof. (*...*) qed.

(* Step 2 *)
lemma Step_Game1_Game2 &m (A <: Adversary) :
  `| Pr[Game1(A).main() @ &m : res] - 1%r / 2%r |
  =
  `| Pr[Game2( A2(A) ).main() @ &m : res] - 1%r / 2%r |.
proof. (*...*) qed.
```

Listing 4.30: First and Second Step in Game-Playing Security Proof

Commencing with `Step_Game1_Game2`, we show that this lemmas indeed corresponds to the security proof's second step by deconstructing it as follows. First, consider `Game1(A).main()` and `Game2( A2(A) ).main()`. As mentioned before, these `main()` procedures contain the actual translations of the games that their modules formalize. Furthermore, `Game1(A)` and `Game2( A2(A) )` denote the instantiations of `Game1` with `A` and, respectively, `Game2` with `A2(A)`. Similarly, `A2(A)` signifies the instantiation of module `A2` with `A`, where `A2` is the module defined in Listing 4.28;

[8]In EasyCrypt, a memory assigns values to all global variables declared in the modules specified by the currently considered script; each procedure call, e.g., `A.guess(sd, b)`, is carried out with respect to such a memory. As such, memories essentially formalize the (global) context in which entities such as adversaries execute.

certainly, this implies that `Game2( A2(A) ).main()` precisely corresponds to the reduction given in Figure 3.10. Combining these observations with the fact that `res` refers to the output of the considered procedure, we deduce that the statements `Pr[Game1(A).main() @ &m : res]` and `Pr[Game2( A2(A) ).main() @ &m : res]` accordingly denote the probability that executing `Game1(A).main()` and `Game2( A2(A) ).main()` in memory `&m` returns `true`. Ergo, accounting for the universal quantification over the valid memories, it follows that these probability statements respectively correspond to $\Pr\left[\text{Game}_{\mathcal{A}}^{1} = 1\right]$ and $\Pr\left[\text{Game}_{\mathcal{R}^{\mathcal{A}}}^{2} = 1\right]$. In particular, these correspondences are valid since, in these latter statements, the global context in which the game and adversary execute (or, in EasyCrypt's terminology, the memory) is left abstract; alternatively stated, $\Pr\left[\text{Game}_{\mathcal{A}}^{1} = 1\right]$ and $\Pr\left[\text{Game}_{\mathcal{R}^{\mathcal{A}}}^{2} = 1\right]$ implicitly perform universal quantification over these contexts. At last, considering ``` `| | ``` constitutes the absolute value operator and `1%r / 2%r` denotes $\frac{1}{2}$, we conclude that lemma `Step_Game1_Game2` accurately formalizes the desired result of the security proof's second step.

Resembling the proof of the second step in Section 3.2.2, the proof of lemma `Step_Game1_Game2` is predominantly concerned with showing that the information provided to an adversary against the first game is identical between a run of the first game and a run of the considered reduction. Particularly, this involves proving that the `cmu` from `Game1(A).main()` invariably equals the `cmu'` from `Game2( A2(A) ).main()`. Comparing the concrete executions of these game formalizations, we see this is indeed the case. Namely, `Game1` computes `cmu` by applying the `scaleRp2R2t` operator to some value constructed in the initial part of the game; contrarily, `Game2` obtains `cmu` by using the `scaleRp2Rppq` operator on this same value. Nullifying this difference, the reduction adversary `A2(A)` applies the `scaleRppq2R2t` operator to the `cmu` it receives from `Game2`, producing `cmu'`. Therefore, it suffices to show that for any `x` of type `Rp`, `scaleRp2R2t x = scaleRppq2R2t (scaleRp2Rppq x)` holds. Indeed, the proof of `Step_Game1_Game2` employs a lemma that states this property. In turn, the verification of this latter lemma depends on numerous properties, primarily concerning the relevant types and operators.

Following a similar line of reasoning, we ascertain that lemma `Step_Distinguish_Game0_Game1_GMLWR` correctly denotes the desired result of the security proof's first step. In particular, we observe that `GMLWR( B0(A) ).main(true)` precisely matches the reduction defined in Figure 3.9 with $u = 1$; likewise, `GMLWR( B0(A) ).main(false)` exactly formalizes this reduction with $u = 0$. As such, the probability statements in `Step_Distinguish_Game0_Game1_GMLWR` accordingly satisfy the correspondences below.

`Pr[Game0(A).main() @ &m : res]`	$\cong$	$\Pr\left[\text{Game}_{\mathcal{A}}^{0} = 1\right]$
`Pr[Game1(A).main() @ &m : res]`	$\cong$	$\Pr\left[\text{Game}_{\mathcal{A}}^{1} = 1\right]$
`Pr[GMLWR( B0(A) ).main(true) @ &m : res]`	$\cong$	$\Pr\left[\text{Game}_{\mathcal{B}_0^{\mathcal{A}},l,\mu,q,p}^{\text{GMLWR}}(1) = 1\right]$
`Pr[GMLWR( B0(A) ).main(false) @ &m : res]`	$\cong$	$\Pr\left[\text{Game}_{\mathcal{B}_0^{\mathcal{A}},l,\mu,q,p}^{\text{GMLWR}}(0) = 1\right]$

From these correspondences, it immediately follows that lemma `Step_Distinguish_Game0_Game1_GMLWR` accurately formalizes the desired result of the security proof's first step.

In contrast to the proof of lemma `Step_Game1_Game2`, the proof of `Step_Distinguish_Game0_Game1_GMLWR` does not employ any properties of performed (sequences of) operations. Namely, due to the construction of reduction adversary `B0(A)`, `GMLWR( B0(A) ).main(true)` and `GMLWR( B0(A) ).main(false)` are semantically equivalent to `Game1(A).main()` and `Game0(A).main()`, respectively. That is, in `GMLWR( B0(A) ).main(true)`, `B0(A)` effectively performs the same operations as `Game1(A).main()`; analogously, in `GMLWR( B0(A) ).main(false)`, `B0(A)` effectively carries out the same operations as `Game0(A).main()`. In fact, the only difference between these executions is that `B0(A)` gets its initial values from `GMLWR`, while `Game0(A).main()` and `Game1(A).main()` acquire these values themselves.

Nevertheless, if `u` equals `true`, then the values given to `B0(A)` are obtained identically to the values in `Game1(A).main()`; similarly, if `u` equals `false`, then the values given to `B0(A)` are acquired in the same manner as the values in `Game0(A).main()`. Concluding, it suffices for the proof of `Step_Distinguish_Game0_Game1_GMLWR` to show these semantic equivalences; certainly, this is precisely the approach that the proof takes.

As alluded to before, after formally verifying the security proof's game-playing steps, the remainder of the formal verification effort for Saber.PKE's IND-CPA security comprises the formalization and (formal) verification of the equivalence between $\mathrm{Game}^{\text{IND-CPA}}_{\mathcal{A},\text{Saber.PKE}}$ and $\mathrm{Game}^{0}_{\mathcal{A}}$, the invariable winning probability of any adversary against $\mathrm{Game}^{0}_{\mathcal{A}}$, and, ultimately, the security theorem. Regarding the first of these remaining endeavors, Listing 4.31 contains the lemma that formalizes the equivalence between $\mathrm{Game}^{\text{IND-CPA}}_{\mathcal{A},\text{Saber.PKE}}$ and $\mathrm{Game}^{0}_{\mathcal{A}}$. Technically, this lemma expresses that for any adversary, the advantage against the former game is equal to the advantage against the latter game; nevertheless, due to the binary output domain of these games, this is tantamount to stating that the games are equivalent. The reason for choosing the advantage-based formalization is that, in this form, the lemma is apter for the formal verification of the final security theorem.

```
(* Saber's INDCPA == Game0 *)
lemma Equivalence_SaberINDCPA_Game0 &m (A <: Adversary) :
  `| Pr[CPA(Saber_PKE_Scheme, A).main() @ &m : res] - 1%r / 2%r |
  =
  `| Pr[Game0(A).main() @ &m : res] - 1%r /2%r |.
proof. (*...*) qed.
```

Listing 4.31: Equivalence Between $\mathrm{Game}^{\text{IND-CPA}}_{\mathcal{A},\text{Saber.PKE}}$ and $\mathrm{Game}^{0}_{\mathcal{A}}$

The verification of `Equivalence_SaberINDCPA_Game0` is rather trivial. Specifically, by inlining the procedures of `Saber_PKE_Scheme` in `CPA(Saber_PKE_Scheme, A).main()`, we obtain a procedure that is semantically equivalent to `Game0(A).main()`[9]; as a result, `CPA(Saber_PKE_Scheme, A).main()` and `Game0(A).main()` can be used interchangeably. Naturally, this directly implies that `Pr[CPA(Saber_PKE_Scheme, A).main() @ &m : res]` is equal to `Pr[Game0(A).main() @ &m : res]`, which is sufficient for proving the considered claim.

Penultimately, the final endeavor preceding the formal verification of the security theorem concerns the formal verification of the invariable winning probability of any adversary against $\mathrm{Game}^{4}_{\mathcal{A}}$; Listing 4.32 presents the lemmas employed in this endeavor.

```
(* Auxiliary_Game Analysis *)
lemma Aux_Prob_Half &m (A <: Adversary) :
  Pr[Auxiliary_Game(A).main() @ &m : res]  = 1%r / 2%r.
proof. (*...*) qed.

(* Pr[Game4(A) = 1] == Pr[Auxiliary_Game(A) = 1] *)
lemma Equal_Prob_Game4_Aux &m (A <: Adversary) :
  Pr[Game4(A).main() @ &m : res] = Pr[Auxiliary_Game(A).main() @ &m : res].
proof. (*...*) qed.

(* Game4 Analysis *)
lemma Game4_Prob_Half &m (A <: Adversary) :
  Pr[Game4(A).main() @ &m : res] = 1%r / 2%r.
proof. (*...*) qed.
```

Listing 4.32: Analysis of $\mathrm{Game}^{4}_{\mathcal{A}}$

[9]In particular, as aforementioned, the polymorphic encoding and decoding operators utilized in `Game0(A).main()` axiomatically equal their monomorphic analogs used in the procedures of `Saber_PKE_Scheme`.

As can be extracted from this listing, `Auxiliary_Game` is indeed employed in the manner described earlier. Namely, first, lemma `Aux_Prob_Half` asserts that for any combination of a valid memory and an adversary of type `Adversary`, the winning probability against `Auxiliary_Game` invariably equals $\frac{1}{2}$. Subsequently, the `Equal_Prob_Game4_Aux` lemma states that the winning probability against `Auxiliary_Game` moreover equals the winning probability against `Game4` for the same memory and adversary. Naturally, from these lemmas, it trivially follows that the winning probability against `Game4` also invariably equals $\frac{1}{2}$, irrespective of the considered memory and adversary. This latter property is explicitly formalized by the final lemma in Listing 4.32, i.e., lemma `Game4_Prob_Half`.

Finally, harnessing the results obtained hitherto, the formal verification effort for Saber.PKE's IND-CPA security culminates with the formalization and (formal) verification of the corresponding security theorem. The ensuing listing, i.e., Listing 4.33, provides the lemmas relevant to this culminating endeavor.

```
(* Intermediate Result *)
lemma Difference_Game1_Game4_XMLWR &m (A <: Adversary):
  `| Pr[Game1(A).main() @ &m : res] - Pr[Game4( A3(A2(A)) ).main() @ &m : res] |
  =
  `| Pr[XMLWR( B1(A3(A2(A))) ).main(true) @ &m : res] - Pr[XMLWR( B1(A3(A2(A)))
    ).main(false) @ &m : res] |.
proof. (*...*) qed.

(* Final Result (Security Theorem) *)
lemma Saber_INDCPA_Security_Theorem &m (A <: Adversary) :
  `| Pr[CPA(Saber_PKE_Scheme, A).main() @ &m : res] - 1%r / 2%r |
  <=
  `| Pr[GMLWR( B0(A) ).main(true) @ &m : res] - Pr[GMLWR( B0(A) ).main(false) @ &m :
    res] |
  +
  `| Pr[XMLWR( B1(A3(A2(A))) ).main(true) @ &m : res] - Pr[XMLWR( B1(A3(A2(A)))
    ).main(false) @ &m : res] |.
proof. (*...*) qed.
```

Listing 4.33: Intermediate Result and Security Theorem

In this listing, the initial lemma, `Difference_Game1_Game4_XMLWR`, corresponds to the intermediate result deduced immediately prior to the derivation of the security theorem in Section 3.2.2; specifically, this lemma states that, given any valid memory `&m` and adversary `A` against `Game1`, there exists adversaries against `Game4` and `XMLWR` such that the difference in winning probabilities between the adversaries against `Game1` and `Game4` is equal to the advantage of the adversary against `XMLWR`. Certainly, these existing adversaries against `Game4` and `XMLWR` can be constructed from an adversary against `Game1` through the formalized reduction adversaries. However, in order to utilize the previously verified lemmas regarding the security proof's steps, these adversaries should not directly be constructed as `A3(A)` and `B1(A)`; rather, the adversaries against `Game4` and `XMLWR` should respectively be constructed as `A3(A2(A))` and `B1(A3(A2(A)))`.

The proof of lemma `Difference_Game1_Game4_XMLWR` ensues as follows. First, by lemma `Game4_Prob_Half`, `Pr[Game4( A3(A2(A)) ).main() @ &m : res]` is replaced by `1%r / 2%r`. Afterward, applying lemma `Step_Game1_Game2`, the resulting left-hand side of the equality, i.e., `` `|Pr[Game1(A).main() @ &m : res] - 1%r / 2%r| ``, is rewritten as `` `|Pr[Game2( A2(A) ).main() @ &m : res] - 1%r / 2%r| ``. Subsequently, the proof utilizes a lemma that, although not explicitly discussed here, is analogous to lemma `Step_Game1_Game2`; however, instead of the step between `Game1` and `Game2`, it concerns the step between `Game2` and `Game3`. As such, by this lemma, `` `|Pr[Game2( A2(A) ).main() @ &m : res] - 1%r / 2%r| `` is replaced with `` `|Pr[Game3( A3(A2(A)) ).main() @ &m : res] - 1%r / 2%r| ``. Lastly, the `1%r / 2%r` is exchanged for `Pr[Game4( A3(A2(A)) ).main() @ &m : res]`, reversing the initial replacement. At this point, the initial statement of lemma `Difference_Game1_Game4_XMLWR` has been transformed in one that, concerning adversary `A3(A2(A))`, precisely matches the `Step_Distinguish_Game3_Game4_XMLWR` lemma. As a result, direct application of this latter lemma concludes the formal verification of `Difference_Game1_Game4_XMLWR`.

Utilizing the above-verified `Difference_Game1_Game4_To_XMLWR` lemma, the formal verification of the final security theorem, i.e., lemma `Saber_INDCPA_Security_Theorem`, proceeds in a manner that closely resembles the corresponding derivation in Section 3.2.2. Foremost, by lemma `Equivalence_SaberINDCPA_Game0`, the left-hand side of the original inequality is replaced by `` `|Pr[Game0(A).main() @ &m : res] - 1%r /2%r|``. Afterward, employing lemma `Game4_Prob_Half` instantiated with adversary `A3(A2(A))`, `1%r / 2%r` is rewritten to `Pr[Game4( A3(A2(A)) ).main() @ &m : res]`. Following, from the triangle inequality for real numbers, the proof deduces that the current left-hand side of the inequality is less than or equal to the sum of `` `|Pr[Game0(A).main() @ &m : res] - Pr[Game1(A).main() @ &m : res]|`` and `` `|Pr[Game1(A).main() @ &m : res] - Pr[Game4( A3(A2(A)) ).main() @ &m : res]|``. In turn, utilizing the lemmas that correspond to the security proof's first step and the intermediate result, this sum is verified to be less than or equal to the sum in the right-hand side of the original inequality. At last, the veracity of the security theorem follows from the transitivity of `<=`.

Altogether, in the formal verification endeavor regarding Saber.PKE's security property, we have formally verified that if GMLWR and XMLWR are hard, Saber.PKE is IND-CPA secure; furthermore, the suitability of GMLWR and XMLWR as hardness assumptions has been formally substantiated through the formal verification of corresponding ROM proofs that relate these assumptions to MLWR.

4.2.3 Correctness

Advancing from the formal verification of Saber.PKE's IND-CPA security, we presently discuss the formal verification of the scheme's other desired property, i.e., its correctness property. As can be extracted from Chapter 3, the security and correctness proofs are, apart from the considered general context and Saber.PKE's original specification, entirely distinct; in turn, the corresponding formal verification endeavors are as well. As such, the preliminaries for the formal verification effort concerning Saber.PKE's correctness, and hence for the ensuing discussion, exclusively comprise the formalizations of the general context and Saber.PKE's specification, respectively addressed in Section 4.1 and Section 4.2.1.

The imminent discussion follows a structure analogous to that of Section 3.2.3. More precisely, we first discuss the formalization of Saber.PKEA and the (formal) verification of the equivalence between Saber.PKE and Saber.PKEA. Afterward, we present the formalizations of the utilized correctness definitions. Lastly, we explicate the formal verification of the correctness analysis provided in the final part of Section 3.2.3.

Alternative Specification of Saber.PKE

Akin to Saber.PKE, Saber.PKEA is a PKE scheme; as such, we formalize Saber.PKEA utilizing the same module type we used for `Saber_PKE_Scheme`. That is, the formalization of Saber.PKEA, `Saber_PKE_Scheme_Alt`, is given the module type `Scheme`. Consequently, this formalization must implement a key generation procedure `kg`, an encryption procedure `enc`, and a decryption procedure `dec`. In agreement with the fact that Saber.KeyGenA precisely matches Saber.KeyGen, the `kg` procedure of `Saber_PKE_Scheme_Alt` is identical to that of `Saber_PKE_Scheme`; contrarily, the other two procedures of `Saber_PKE_Scheme_Alt` deviate from their counterparts of `Saber_PKE_Scheme`, congruent with the differences between the corresponding algorithms of Saber.PKEA and Saber.PKE. For this reason, the implementation of the `kg` procedure from `Saber_PKE_Scheme_Alt` is not explicitly shown here, yet the implementations of the other two procedures from `Saber_PKE_Scheme_Alt` are. Covering the first of these two implementations, Listing 4.34 provides the implementation of `enc`.

```
module Saber_PKE_Scheme_Alt : Scheme = {
   proc kg() : pkey * skey = {
      (*...*)
   }

   proc enc(pk: pkey, m: plaintext) : ciphertext = {
      var pk_dec: seed * Rp_vec;
      var m_dec: R2;
      var sd: seed;
      var _A: Rq_mat;
      var s': Rq_vec;
      var b, b': Rp_vec;
      var bq: Rq_vec;
      var v': Rq;
      var cm: R2t;

      m_dec <- m_decode m;
      pk_dec <- pk_decode_s pk;
      sd <- pk_dec.`1;
      b <- pk_dec.`2;

      _A <- gen sd;
      s' <$ dsmallRq_vec;
      b' <- scaleRqv2Rpv ((trmx _A) *^ s' + h);
      bq <- scaleRpv2Rqv b;
      v' <- (dotp bq s') + (upscaleRq h1 (eq - ep));
      cm <- scaleRq2R2t (v' + (scaleR22Rq m_dec));

      return c_encode_s (cm, b');
   }

   proc dec(sk: skey, c: ciphertext) : plaintext option = {

   }
}.
```

Listing 4.34: Saber.EncA's Specification

Apart from the necessary encoding and decoding operations regarding the abstract cryptographic types, the implementation of `enc` is a relatively straightforward translation of Saber.EncA's specification. Namely, considering all of the material discussed hitherto, this implementation does not comprise any unprecedented concepts or operators; that is, except for the utilized `upscaleRq` operator. However, the definition and purpose of this operator are trivially derived from its application in the implementation. Particularly, `upscaleRq` multiplies each coefficient of its first argument, an element of type `Rq`, by an integral power-of-two; indeed, the exponent of this power-of-two is provided through the operator's second argument. Moreover, the output of this operator is an element of type `Rq`. As such, `upscaleRq h1 (eq - ep)` correctly formalizes $2^{\epsilon_q - \epsilon_p} \cdot h_1 = \frac{q}{p} \cdot h_1$.

Completing the formalization of Saber.PKEA, Listing 4.35 presents the implementation of the `dec` procedure from `Saber_PKE_Scheme_Alt`.

```
module Saber_PKE_Scheme_Alt : Scheme = {
   proc kg() : pkey * skey = {
      (*...*)
   }

   proc enc(pk: pkey, m: plaintext) : ciphertext = {
      (*...*)
   }

   proc dec(sk: skey, c: ciphertext) : plaintext option = {
      var c_dec: R2t * Rp_vec;
      var cm: R2t;
      var cmq: Rq;
      var b': Rp_vec;
      var bq': Rq_vec;
      var v: Rq;
      var s: Rq_vec;
      var m': R2;

      c_dec <- c_decode_s c;
      s <- sk_decode_s sk;
      cm <- c_dec.`1;
      b' <- c_dec.`2;

      cmq <- scaleR2t2Rq cm;
      bq' <- scaleRpv2Rqv b';
      v <- (dotp bq' s) + (upscaleRq h1 (eq - ep));
      m' <- scaleRq2R2 (v - cmq + (upscaleRq h2 (eq - ep)));

      return Some (m_encode m');
   }
}.
```

Listing 4.35: Saber.DecA's Specification

Similarly to the formalization of Saber.EncA, `dec` is a rather trivial translation of Saber.DecA's specification; in fact, with the introduction of the `upscaleRq` operator above, this implementation does not contain any novel concepts or operators that require additional elaboration.

Finally, concerning the formal verification of the equivalence between Saber.PKE and Saber.PKEA, Listing 4.36 defines the relevant lemmas. Specifically, for each pair of analogous procedures between `Saber_PKE_Scheme` and `Saber_PKE_Scheme_Alt`, this listing contains a lemma stating their equivalence; indeed, this is consistent with the definition of scheme equivalence discussed in Section 3.2.3. Namely, in this context, two schemes are equivalent if each algorithm in one scheme is equivalent to exactly one algorithm in the other scheme; two algorithms are equivalent if they have identical in and output domains and, given the same input, they produce identical output.

```
(* Equivalence of Key Generation *)
lemma Equivalence_kg:
  equiv[Saber_PKE_Scheme.kg ~ Saber_PKE_Scheme_Alt.kg : true ==> ={res}].
proof. (*...*) qed.

(* Equivalence of Encryption *)
lemma Equivalence_enc:
  equiv[Saber_PKE_Scheme.enc ~ Saber_PKE_Scheme_Alt.enc : ={pk, m} ==> ={res}].
proof. (*...*) qed.

```

```
(* Equivalence of Decryption *)
lemma Equivalence_dec:
  equiv[Saber_PKE_Scheme.dec ~ Saber_PKE_Scheme_Alt.dec : ={sk, c} ==> ={res}].
proof. (*...*) qed.
```

Listing 4.36: Equivalence between Saber.PKE and Saber.PKEA

Exemplifying the interpretation of the lemmas in Listing 4.36, we elaborate on the lemma that denotes the equivalence between the encryption procedures, i.e., lemma `Equivalence_enc`. In this lemma, although applicable to more general situations as well, `equiv` is utilized to state the desired equivalence. More precisely, inside the square brackets belonging to `equiv`, the part preceding the colon indicates the procedures that the statement concerns; as desired, in lemma `Equivalence_enc`, these procedures are `Saber_PKE_Scheme.enc` and `Saber_PKE_Scheme_Alt.enc`. Furthermore, the part succeeding the colon defines the conditions on the initial and resulting memories under which the equivalence holds. In this case, `={pk, m} ==> ={res}` conveys that, for any combination of valid memories such that the provided public key and message are equal between these memories, executing the considered procedures in their respective memories entails an equal probability for both procedures to output any specific value. More concretely, consider any two valid memories: one for `Saber_PKE_Scheme.enc`, the other for `Saber_PKE_Scheme_Alt.enc`. Additionally, let the values assigned to `pk` and `m` by the former memory be equal to, respectively, the values assigned to `pk` and `m` by the latter memory. Then, when executing each procedure in their respective memories, the probability of `Saber_PKE_Scheme.enc` returning `true` is precisely equal to the probability of `Saber_PKE_Scheme_Alt.enc` returning `true`; analogously, this also holds for the output value `false`.

Comparing the interpretation of the lemmas in Listing 4.36 to the definition of (algorithm) equivalence employed in Section 3.2.3, we see that these lemmas merely formalize the requirement stating that, given the same input, the collated algorithms must invariably produce identical output. In particular, the lemmas do not formalize the requirement denoting that the algorithms' input and output domains must be identical. Nevertheless, albeit not formally verifiable in EasyCrypt, the conformance to this requirement is trivially confirmed through manual inspection. Namely, this conformance directly follows from the fact that all procedures have the same parameter types in the same order as their counterparts from the other scheme.

The proofs of the above-discussed equivalence lemmas proceed as expected based on the discussion in Section 3.2.3. Particularly, the proof of `Equivalence_kg` directly verifies the statement of this lemma by utilizing that `Saber_PKE_Scheme.kg` and `Saber_PKE_Scheme_Alt.kg` are identical. Furthermore, the proofs of `Equivalence_enc` and `Equivalence_dec` verify the statements of these lemmas by showing the equivalence between the computations performed by the collated procedures.

Correctness Definitions

Employing the established formalizations of Saber.PKE and Saber.PKEA, we formalize the relevant correctness definitions, i.e., standard correctness and FO-correctness, for both schemes. Furthermore, we formally verify that standard correctness and FO-correctness with respect to Saber.PKE are equivalent to, respectively, standard correctness and FO-correctness with respect to Saber.PKEA. Indeed, these equivalences enable the remainder of the formal verification effort to, for either definition, consider the correctness of Saber.PKEA rather than the correctness of Saber.PKE.

Given all of the previously discussed material, it is a relatively straightforward endeavor to formalize the correctness definitions defined in Figure 3.13 and Figure 3.14. Nevertheless, rather than constructing a separate formalization for Saber.PKE and Saber.PKEA, we establish a single, more abstract formalization per correctness definition. Specifically, for each of standard correctness and FO-correctness, we construct a single module that is parameterized by another module of type `Scheme`; indeed, since the `Scheme` module type is designed for the formalization of

PKE schemes, this essentially formalizes the correctness definitions concerning a general PKE scheme, i.e., $\text{PProg}^{\text{STDCOR}}_{\text{PKE}}$ and $\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{PKE}}$. As such, we prevent unnecessary duplication while retaining the capability of referring to the concrete correctness definitions for Saber.PKE and Saber.PKEA. Listing 4.37 comprises this formalization regarding standard correctness (for a general PKE scheme).

```
module Correctness_Standard (S : Scheme) = {
   proc main(m : plaintext) : bool = {
      var m': plaintext option;
      var c: ciphertext;
      var pk: pkey;
      var sk: skey;

      (pk, sk) <@ S.kg();
      c <@ S.enc(pk, m);
      m' <@ S.dec(sk, c);

      return (Some m = m');
   }
}.
```

Listing 4.37: Standard Correctness Definition (i.e., $\text{PProg}^{\text{STDCOR}}_{\text{PKE}}$)

Certainly, instantiating this module with `Saber_PKE_Scheme` and `Saber_PKE_Scheme_Alt` produces the formalizations of $\text{PProg}^{\text{STDCOR}}_{\text{Saber.PKE}}$ and $\text{PProg}^{\text{STDCOR}}_{\text{Saber.PKEA}}$, respectively. The equivalence between these formalizations is formally verified by employing a lemma similar to those presented in Listing 4.36; this lemma is shown in the ensuing listing.

```
lemma Equivalence_Correctness_Standard_Orig_Alt :
  equiv[Correctness_Standard(Saber_PKE_Scheme).main ~
    Correctness_Standard(Saber_PKE_Scheme_Alt).main : ={m} ==> ={res}].
proof. (*...*) qed.
```

Listing 4.38: Equivalence Between $\text{PProg}^{\text{STDCOR}}_{\text{Saber.PKE}}$ and $\text{PProg}^{\text{STDCOR}}_{\text{Saber.PKEA}}$

The interpretation of `Equivalence_Correctness_Standard_Orig_Alt` is analogous to the interpretation of the formerly explicated equivalence lemmas. Furthermore, the proof of this lemma is exclusively contingent on the previously verified equivalences between the procedures of `Saber_PKE_Scheme` and `Saber_PKE_Scheme_Alt`.

As elaborated on in Section 3.2.3, contrariwise to standard correctness, FO-correctness is defined with respect to an adversary; more precisely, FO-correctness considers an adversary that, given a public key and a secret key, produces a message. As with former adversaries, we formalize this adversary by means of a distinct module type; this module type is defined in Listing 4.39.

```
module type Adv_Cor = {
   proc choose(pk : pkey, sk : skey) : plaintext
}.
```

Listing 4.39: Module Type for Adversaries Against $\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{PKE}}$

Then, utilizing `Adv_Cor` to formalize $\mathcal{A}$, we construct the module that serves as the formalization of FO-correctness for a general PKE scheme; Listing 4.40 provides this module.

```
module Correctness_Game (S : Scheme, A : Adv_Cor) = {
   proc main() : bool = {
      var m: plaintext;
      var m': plaintext option;
      var c: ciphertext;
      var pk: pkey;
      var sk: skey;

      (pk, sk) <@ S.kg();
      m <@ A.choose(pk, sk);
      c <@ S.enc(pk, m);
      m' <@ S.dec(sk, c);

      return (Some m = m');
   }
}.
```

Listing 4.40: FO-Correctness Definition (i.e., $\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{PKE}}$)

Similarly to `Correctness_Standard`, the module in this listing can be instantiated with `Saber_PKE_Scheme` and `Saber_PKE_Scheme_Alt` to produce the formalizations of FO-correctness for Saber.PKE and, respectively, Saber.PKEA. Specifically, `Correctness_Game(Saber_PKE_Scheme, A)` and `Correctness_Game(Saber_PKE_Scheme_Alt, A)` accordingly formalize $\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{Saber.PKE}}$ and $\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{Saber.PKEA}}$, where `A` represents $\mathcal{A}$.

Lastly, we formally verify the equivalence between the formalizations of $\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{Saber.PKE}}$ and $\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{Saber.PKEA}}$ through a lemma that is analogous to the lemma in Listing 4.38. In particular, this lemma states that for any module `A` of type `Adv_Cor`, `Correctness_Game(Saber_PKE_Scheme, A)` is equivalent to `Correctness_Game(Saber_PKE_Scheme_Alt, A)`. Moreover, akin to the proof of `Equivalence_Correctness_Standard_Orig_Alt`, the proof of this lemma solely depends on the equivalences between the procedures from `Saber_PKE_Scheme` and `Saber_PKE_Scheme_Alt`.

Correctness Equivalence and Error Expression

In the correctness-related formal verification effort, we have hitherto formalized Saber.PKEA and the two relevant correctness definitions, i.e., standard correctness and FO-correctness. Furthermore, we have formally verified that the correctness definitions in the context of Saber.PKE are equivalent to their analogs in the context of Saber.PKEA. Leveraging these established formalizations and results, the remainder of this formal verification effort continues to resemble the manual analysis carried out in Section 3.2.3. Namely, first, we formalize the error expression considered by Saber's script to exhaustively compute the correctness of Saber.PKE. Then, we formalize $\text{PProg}^{\text{COR}}$, i.e., the probabilistic program that almost represents the computation carried out by Saber's script, solely deviating from an accurate representation of this computation regarding the distribution of the considered matrix $\mathbf{A}$. Afterward, we formally verify that this probabilistic program is equivalent to both standard correctness and FO-correctness for Saber.PKEA; indeed, by the transitivity of equivalences, this shows that these correctness definitions are additionally equivalent in the context of Saber.PKE. Penultimately, we formalize $\text{PProg}^{\delta\text{COR}}$, the probabilistic program that accurately represents the computation carried out by Saber's script. Lastly, under the assumption that `gen` is a random oracle, we formally verify that in the context of Saber.PKE, the considered correctness definitions are equivalent to $\text{PProg}^{\delta\text{COR}}$; certainly, this proves that if the output distribution of (the instantiation of) `gen` (practically) equals the uniform distribution over the domain of $\mathbf{A}$, the correctness of Saber.PKE is (almost) accurately computed by Saber's script.

First and foremost, we reiterate the definitions of the error expression and error terms provided in Section 3.2.3. Specifically, the error expression is defined as follows.

$$\mathbf{err}_{\mathbf{b}'_q}^T \cdot \mathbf{s} - \mathbf{err}_{\mathbf{b}_q}^T \cdot \mathbf{s}' - \mathrm{err}_{c_{mq}}$$

Here, $\mathbf{err}_{\mathbf{b}'_q}$, $\mathbf{err}_{\mathbf{b}_q}$, and $\mathrm{err}_{c_{mq}}$ denote the error terms; the definitions of these terms are repeated below.

$$\begin{aligned}
\mathbf{err}_{\mathbf{b}'_q} &= \mathbf{b}'_q - \mathbf{A}^T \cdot \mathbf{s}' = \lfloor \lfloor \mathbf{A}^T \cdot \mathbf{s}' + \mathbf{h} \rfloor_{q \to p} \rfloor_{p \to q} - \mathbf{A}^T \cdot \mathbf{s}' \\
\mathbf{err}_{\mathbf{b}_q} &= \mathbf{b}_q - \mathbf{A} \cdot \mathbf{s} = \lfloor \lfloor \mathbf{A} \cdot \mathbf{s} + \mathbf{h} \rfloor_{q \to p} \rfloor_{p \to q} - \mathbf{A} \cdot \mathbf{s} \\
\mathrm{err}_{c_{mq}} &= c_{mq} - (v' + \lfloor m \rfloor_{2 \to q}) + \frac{q}{4 \cdot t} = \lfloor \lfloor v' + \lfloor m \rfloor_{2 \to q} \rfloor_{q \to 2 \cdot t} \rfloor_{2 \cdot t \to q} - (v' + \lfloor m \rfloor_{2 \to q}) + \frac{q}{4 \cdot t}
\end{aligned}$$

In both the error expression and the error terms, all variables refer to the identically denoted artifacts considered in Saber.PKEA.

Illustrating the manner in which the above-reiterated error terms are formalized, we examine the formalizations of $\mathbf{err}_{\mathbf{b}_q}$ and $\mathbf{err}_{\mathbf{b}'_q}$; these formalizations are provided in Listing 4.41.

```
op error_bq (_A : Rq_mat) (s : Rq_vec) : Rq_vec =
  (scaleRpv2Rqv (scaleRqv2Rpv (_A *^ s + h))) - (_A *^ s).

op error_bq' (_A : Rq_mat) (s': Rq_vec) : Rq_vec =
  (scaleRpv2Rqv (scaleRqv2Rpv ((trmx _A) *^ s' + h))) - ((trmx _A) *^ s').
```

Listing 4.41: Error Terms $\mathbf{err}_{\mathbf{b}_q}$ and $\mathbf{err}_{\mathbf{b}'_q}$

As this listing demonstrates, we formalize the error terms as parameterized operators. For intelligibility purposes, the parameters of these operators are denominated identically to the artifacts with which they are intended to be instantiated. For example, the formalization of $\mathbf{err}_{\mathbf{b}_q}$, i.e., `error_bq`, is defined with respect to parameters `_A` of type `Rq_mat` and `s` of type `Rq_vec`. Indeed, these parameters are expected to be instantiated with, respectively, the `_A` and `s` artifacts from `Saber_PKE_Scheme_Alt`; since these artifacts accordingly formalize $\mathbf{A}$ and $\mathbf{s}$ from Saber.PKEA, this is consistent with the fact that, excluding constants, $\mathbf{err}_{\mathbf{b}_q}$ is only dependent on this $\mathbf{A}$ and $\mathbf{s}$. Provided with concrete values for these parameters, `error_bq` is defined as `(scaleRpv2Rqv (scaleRqv2Rpv (_A *^ s + h))) - (_A *^ s)`, which formalizes $\lfloor \lfloor \mathbf{A} \cdot \mathbf{s} + \mathbf{h} \rfloor_{q \to p} \rfloor_{p \to q} - \mathbf{A} \cdot \mathbf{s}$. Certainly, as substantiated by the definitions reiterated above, this precisely matches the expression corresponding to $\mathbf{err}_{\mathbf{b}_q}$. A comparable analysis shows that the other operator in Listing 4.41, i.e., `error_bq'`, accurately formalizes the error term $\mathbf{err}_{\mathbf{b}'_q}$.

Utilizing the formalizations of the error terms, we formalize the error expression in a similar fashion, i.e., by means of a parameterized operator; this operator is defined in Listing 4.42.

```
op error_expression (_A : Rq_mat) (s : Rq_vec) (s' : Rq_vec) =
  dotp (error_bq' _A s') s - dotp (error_bq _A s) s' - error_cmq_nom_centered _A s s'
```

Listing 4.42: Error Expression

As with `error_bq` and `error_bq'`, `error_expression` is intended to be instantiated with the formalizations of the relevant artifacts from Saber.PKEA. Specifically, this operator is meant to receive the `A`, `s`, and `s'` produced by `Saber_PKE_Scheme_Alt`. Combining this with the preceding discussion, it follows that the first two terms in this operator's definition correctly formalize $\mathbf{err}_{\mathbf{b}_q}^T \cdot \mathbf{s}'$ and $\mathbf{err}_{\mathbf{b}'_q}^T \cdot \mathbf{s}$, respectively. The remaining term, i.e., `error_cmq_nom_centered _A s s'`, formalizes the expression for $\mathrm{err}_{c_{mq}}$ that does not include the message m; that is, it formalizes $\lfloor \lfloor v' \rfloor_{q \to 2 \cdot t} \rfloor_{2 \cdot t \to q} - v' + \frac{q}{4 \cdot t}$, as derived in Section 3.2.3.

Building on the formalization of the error expression, we formalize $\mathrm{PProg}^{\mathrm{COR}}$, the probabilistic program that almost denotes the computation carried out by Saber's script, originally specified in Figure 3.15. As mentioned before, the only difference between PProg$^{\text{COR}}$ and a program that accurately represents the computation performed by Saber's script concerns the distribution of the considered matrix $\mathbf{A}$. Namely, PProg$^{\text{COR}}$ generates $\mathbf{A}$ through gen while, to accurately represent the computation performed by Saber's script, $\mathbf{A}$ must necessarily be uniformly distributed. Indeed, in PProg$^{\text{COR}}$, $\mathbf{A}$ is only uniformly distributed if gen is (assumed to be) a random oracle; in turn, PProg$^{\text{COR}}$ also only accurately represents this computation if gen is (assumed to be) a random oracle. In fact, we formalize and verify this reasoning momentarily. However, before doing so, we address the formalization of PProg$^{\text{COR}}$ and the formal verification of the equivalences between this program, PProg$^{\text{STDCOR}}_{\text{Saber.PKEA}}$, and Game$^{\text{FOCOR}}_{\mathcal{A},\text{Saber.PKEA}}$. To this end, first, Listing 4.43 provides the definition of the module that formalizes PProg$^{\text{COR}}$.

```
module Correctness_PProg = {
   proc main() : bool = {
      var sd: seed;
      var _A : Rq_mat;
      var s, s': Rq_vec;

      sd <$ dseed;
      _A <- gen sd;
      s <$ dsmallRq_vec;
      s' <$ dsmallRq_vec;

      return coeffs_corr_rng (error_expression _A s s');
   }
}.
```

Listing 4.43: PProg$^{\text{COR}}$

Examining the definition of `Correctness_PProg`, we see that the `coeffs_corr_rng` operator is the only novelty introduced in this module; indeed, the remainder of this module is a straightforward translation of PProg$^{\text{COR}}$ that exclusively utilizes formerly explained concepts. As its name suggests, the `coeffs_corr_rng` operator formalizes the coeffs_in_correctness_rng predicate defined in Section 3.2.3. That is, `coeffs_corr_rng` takes an argument of type `Rq` and evaluates to true if and only if all of its argument's coefficients lie between `-q %/ 4` (including) and `q %/ 4` (excluding). In the case of `Correctness_PProg`, the argument given to this operator is `error_expression _A s s'`, an instance of the formalized error expression. As such, `coeffs_corr_rng (error_expression _A s s')` correctly formalizes the return statement of PProg$^{\text{COR}}$.

Employing the formalizations of PProg$^{\text{COR}}$, PProg$^{\text{STDCOR}}_{\text{Saber.PKEA}}$, and Game$^{\text{FOCOR}}_{\mathcal{A},\text{Saber.PKEA}}$, we formally verify the equivalences between PProg$^{\text{COR}}$ and the correctness definitions for Saber.PKEA. In particular, we do so by means of the equivalence lemmas presented in Listing 4.44; certainly, the interpretation of these lemmas is similar to the interpretation of the previously discussed equivalence lemmas.

```
lemma Equivalence_CorrStd_CorrPProg :
  equiv[Correctness_Standard(Saber_PKE_Scheme_Alt).main ~ Correctness_PProg.main :
    true ==> ={res}].
proof. (*...*) qed.

lemma Equivalence_CorrStd_CorrGame (A <: Adv_Cor) :
  equiv[Correctness_Game(Saber_PKE_Scheme_Alt, A).main ~ Correctness_PProg.main :
    true ==> ={res}].
proof. (*...*) qed.
```

Listing 4.44: Equivalences Between PProg$^{\text{STDCOR}}_{\text{Saber.PKEA}}$, Game$^{\text{FOCOR}}_{\mathcal{A},\text{Saber.PKEA}}$, and PProg$^{\text{COR}}$

Conceptually, the process of formally verifying either of these lemmas is quite simplistic; moreover, this process is identical for both lemmas. Namely, we initiate both proofs by verifying that replacing the return expression `Some m' = m` by `coeffs_corr_rng (error_expression _A s s')` produces an equivalent program. Subsequently, we remove all statements, variables, and entities that do not affect the manner in which the artifacts `_A`, `s`, and `s'` are obtained; indeed, this includes the removal of the considered message `m` and, for `Correctness_Game(Saber_PKE_Scheme_Alt, A).main`, the adversary `A`. Since the substitute return statement, i.e., `coeffs_corr_rng (error_expression _A s s')`, exclusively depends on `_A`, `s`, and `s'`, this removal trivially produces an equivalent program. After these transformations, the considered initial program, i.e., either `Correctness_Standard(Saber_PKE_Scheme_Alt).main` or `Correctness_Game(Saber_PKE_Scheme_Alt, A).main`, has become identical to `Correctness_PProg.main`; naturally, this identity directly implies the equivalence between the program resulting from the transformations and `Correctness_PProg.main`. Lastly, because the sequence of performed transformations is formally verified to yield a program that is equivalent to the initial program, it follows that this initial program is equivalent to `Correctness_PProg.main`. Certainly, in the case of lemma `Equivalence_CorrStd_CorrPProg`, this initial program is `Correctness_Standard(Saber_PKE_Scheme_Alt).main`; in the case of lemma `Equivalence_CorrStd_CorrGame`, this initial program is `Correctness_Game(Saber_PKE_Scheme_Alt, A).main`. This completes the formal verification process for both of these lemmas.

Despite its conceptual simplicity, the above-described formal verification process for `Equivalence_CorrStd_CorrPProg` and `Equivalence_CorrGame_CorrPProg` is significantly more complex and strenuous in practice. This intricacy predominantly arises from the necessity of proving a plethora of auxiliary properties regarding the relevant (sequences of) operations. Particularly, in verifying that replacing the return statement in `Correctness_Standard(Saber_PKE_Scheme_Alt).main` and `Correctness_Game(Saber_PKE_Scheme_Alt, A).main` yields an equivalent program, we essentially need to formally verify the majority of the mathematical derivations presented in Section 3.2.3. However, because these derivations intertwine a multitude of different, potentially unconventional operators, the properties required to formally verify their validity are not predefined in EasyCrypt. As such, we are necessitated to both formalize and verify these required properties. Since this endeavor is rather technically involved, it is not suited for explication in the current discussion. Therefore, for further details on the formal verification of the mathematical derivations discussed in Section 3.2.3, refer to the actual code.

Finally, we proceed to the formal verification of the main correctness-related result for Saber.PKE. To reiterate, this result states the following: With respect to both the standard correctness and FO-correctness definitions, Saber.PKE's correctness is a constant probability $1 - \delta \in [0, 1]$ that, under the assumption that `gen` is a random oracle, is accurately computed in Saber's script. Initiating the formal verification effort for this result, we formalize $\mathrm{PProg}^{\delta\mathrm{COR}}$, the probabilistic program that accurately captures the computation performed by Saber's script; this probabilistic program is defined in Figure 3.16. Indeed, unlike $\mathrm{PProg}^{\mathrm{COR}}$, $\mathrm{PProg}^{\delta\mathrm{COR}}$ directly samples $\mathbf{A}$ from $\mathcal{U}(R_q^{l \times l})$, ensuring that $\mathbf{A}$ is uniformly distributed; this is in accordance with the assumed distribution of this matrix in Saber's script. Listing 4.45 provides the formalization of $\mathrm{PProg}^{\delta\mathrm{COR}}$.

```
module Delta_Prob_PProg = {
   proc main() : bool = {
      var sd: seed;
      var _A : Rq_mat;
      var s, s': Rq_vec;

      _A <$ dRq_mat;
      s <$ dsmallRq_vec;
      s' <$ dsmallRq_vec;

      return coeffs_corr_rng (error_expression _A s s');
```

```
    }
};
```

Listing 4.45: $\text{PProg}^{\delta\text{COR}}$

As can be extracted from this listing, `Delta_Prob_PProg` is a straightforward translation of $\text{PProg}^{\delta\text{COR}}$, containing no unprecedented concepts or operators; therefore, no further elaboration is provided for this formalization.

For reasons of completeness, albeit trivially true, we formally verify that $\Pr\left[\text{PProg}^{\delta\text{COR}} = 1\right]$ is actually constant. In particular, we do so by showing that for any valid contexts, this probability is identical. Here, "context" refers to a particular assignment of concrete values to all artifacts that might have an effect on the program's execution. Indeed, in EasyCrypt, this would generally constitute a combination of a memory and concrete module and procedure parameter values. Nevertheless, since $\text{PProg}^{\delta\text{COR}}$ is not defined with respect to any adversaries or (function) parameters, `Delta_Prob_PProg` and its `main` procedure are not defined with respect to any parameters either. As such, the lemma formalizing the property that $\Pr\left[\text{PProg}^{\delta\text{COR}} = 1\right]$ is constant can only differentiate between the considered memories; this lemma is given in Listing 4.46.

```
lemma DeltaProb_Constant &m1 &m2 :
  Pr[Delta_Prob_PProg.main() @ &m1 : res] = Pr[Delta_Prob_PProg.main() @ &m2 : res].
proof. (*...*) qed.
```

Listing 4.46: $\Pr\left[\text{PProg}^{\delta\text{COR}} = 1\right]$ is Constant

Due to the fact that `Delta_Prob_PProg.main` produces all the values on which it operates by itself, the considered memory has no influence on its execution; hence the probability of `Delta_Prob_PProg.main` returning a specific value is trivially constant. Consequently, the proof of this lemma merely requires generic reasoning principles to conclude that this lemma is valid.

Associated with `Delta_Prob_PProg`, Listing 4.47 defines a constant and an axiom formalizing the assumption that $\Pr\left[\text{PProg}^{\delta\text{COR}} = 1\right] = 1-\delta$, where $1-\delta$ represents the correctness value computed by Saber's script for the considered parameter set.

```
const delta_prob : real.

axiom delta_correctness &m :
  Pr[Delta_Prob_PProg.main() @ &m : res] = 1%r - delta_prob.
```

Listing 4.47: $1 - \delta$ Probability Assumption

At this point, the remainder of the formal verification effort primarily entails proving the equivalence between `Correctness_PProg.main` and `Delta_Prob_PProg.main`, i.e., under the assumption that `gen` is a random oracle. Subsequently, together with this equivalence, we can employ the previously verified equivalences to derive that `Correctness_Standard(Saber_PKE_Scheme).main` and `Correctness_Game(Saber_PKE_Scheme, A).main` are equivalent to `Delta_Prob_PProg.main`. In turn, these latter equivalences directly imply that the probability of `Correctness_Standard(Saber_PKE_Scheme).main(m)` and `Correctness_Game(Saber_PKE_Scheme, A).main()` returning `true` is, irrespective of the considered message or adversary, equal to that of `Delta_Prob_PProg.main()`; naturally, due to the `delta_correctness` axiom, this probability precisely equals `1 - delta_prob`.

Concretizing the above-described remainder of the formal verification effort, we first define a variant of `Correctness_PProg` which we will refer to as `Correctness_PProg_RO`. As with `GMLWR_RO` in relation to `GMLWR` (see Section 4.2.2), `Correctness_PProg_RO` is an exact copy of `Correctness_PProg`, solely replacing `gen` by a random oracle[10]; however, in this case, the replacement oracle is a regular non-programmable random oracle. Afterward, formalizing the assumption that `gen` is a random oracle, we introduce an axiom stating the equivalence between `Correctness_PProg.main` and `Correctness_PProg_RO.main`. Following, utilizing a lemma analogous to the preceding equivalence lemmas, we verify the equivalence between `Correctness_PProg_RO.main` and `Delta_Prob_PProg.main`. Indeed, the veracity of this lemma immediately follows from the fact that the oracle's output distribution is uniform, which is precisely what the lemma's proof argues. Penultimately, we formally verify the equivalence between $\text{PProg}^{\delta\text{COR}}$ and the correctness definitions for Saber.PKE; more precisely, we do so through the equivalence lemmas presented in Listing 4.48.

```
lemma Equivalence_CorrStd_DeltaProb :
  equiv[Correctness_Standard(Saber_PKE_Scheme).main ~ Delta_Prob_PProg.main : true
    ==> ={res}].
proof. (*...*) qed.

lemma Equivalence_CorrGame_DeltaProb (A <: Adv_Cor) :
  equiv[Correctness_Game(Saber_PKE_Scheme, A).main ~ Delta_Prob_PProg.main : true
    ==> ={res}].
proof. (*...*) qed.
```

Listing 4.48: Equivalences Between $\text{PProg}^{\text{STDCOR}}_{\text{Saber.PKE}}$, $\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{Saber.PKE}}$, and $\text{PProg}^{\delta\text{COR}}$

As alluded to before, the proofs of these lemmas exclusively require the utilization of the other equivalences verified thus far.

At last, we formalize and verify the final result from Section 3.2.3, i.e., $\Pr\left[\text{PProg}^{\text{STDCOR}}_{\text{Saber.PKE}}(m) = 1\right] = \Pr\left[\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{Saber.PKE}} = 1\right] = 1 - \delta$, by means of the lemmas provided in Listing 4.49.

```
lemma Delta_Correctness_Standard_Original_Scheme &m (msg : plaintext) :
  Pr[Correctness_Standard(Saber_PKE_Scheme).main(msg) @ &m : res] = 1%r - delta_prob.
proof. (*...*) qed.

lemma Delta_Correctness_Game_Original_Scheme &m (A <: Adv_Cor) :
  Pr[Correctness_Game(Saber_PKE_Scheme, A).main() @ &m : res] = 1%r - delta_prob.
proof. (*...*) qed.
```

Listing 4.49: $\Pr\left[\text{PProg}^{\text{STDCOR}}_{\text{Saber.PKE}}(m) = 1\right] = \Pr\left[\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{Saber.PKE}} = 1\right] = 1 - \delta$

The truth of these lemmas directly follows from the equivalences established in Listing 4.48 and the `delta_correctness` axiom; certainly, this is exactly the approach utilized by the respective proofs.

To conclude, in the formal verification effort concerning Saber.PKE's correctness property, we have formally verified that standard correctness and FO-correctness are equivalent in the context of Saber.PKE. Moreover, irrespective of the utilized definition, Saber.PKE's correctness constitutes a constant probability that, assuming `gen` is a random oracle, is accurately computed by Saber's script. Alternatively stated, if the output distribution of (the instantiation of) `gen` (closely) resembles the uniform distribution over $R_q^{l \times l}$, Saber.PKE's correctness is (almost) accurately computed by Saber's script.

[10]The concrete implementation of this random oracle is predefined in EasyCrypt's standard library.

4.3 Demonstration: Proving Lemmas in EasyCrypt

In the preceding discussion on the formal verification endeavor regarding Saber.PKE, we did not elaborate on any concrete proofs corresponding to the employed lemmas. As alluded to before, the principal rationale behind this arises from the fact that the exposition of such esoteric and technical undertakings does not constitute a significant contribution to a discussion with a process-oriented character. Nonetheless, to still provide an impression of EasyCrypt's (lemma-)proving process and the corresponding fundamental concepts and mechanisms, we presently discuss the concrete formal verification of the `Step_Game1_Game2` lemma; originally specified in Listing 4.30, this lemma formalizes the second step of Saber.PKE's game-playing security proof.

This section is structured as follows. Initially, we cover the relevant concepts and mechanisms related to the process of formally verifying lemmas in EasyCrypt. Afterward, employing these concepts and mechanisms, we discuss the concrete proof of the `Step_Game1_Game2` lemma.

4.3.1 Fundamental Concepts and Mechanisms

Preceding the explication of the concrete proof for lemma `Step_Game1_Game2`, we elaborate on several of EasyCrypt's fundamental concepts and mechanisms relevant to the formal verification of lemmas. Foremost, EasyCrypt's *proof engine*, i.e., the component of EasyCrypt that manages the formal verification process for lemmas, is based on the concept of goals. A *goal* comprises two primary components: a *context* and a *conclusion*. Here, the former consists of an ordered set of assumptions; the latter constitutes a single well-defined statement[11]. Considering such a goal, *tactics*, i.e., built-in logical reasoning principles, can be employed to formally verify the goal's conclusion in the goal's context. More precisely, if applicable to the considered goal, a tactic effectively replaces this goal by zero or more (other) goals; by construction, the conjunction of these generated goals' conclusions implies the original goal's conclusion. In case a tactic replaces a goal with precisely zero goals, the tactic is said to *solve* this goal; that is, the tactic proves the truth of this goal, completing its formal verification.

Exemplifying the general representation of goals during the process of proving a lemma in EasyCrypt, Listing 4.50 depicts (EasyCrypt's representation of) the initial goal induced by `lemma scaleRp2Rppq2R2t_comp (p : Rp) : scaleRp2R2t p = scaleRppq2R2t (scaleRp2Rppq p)`, a lemma that specifies a property required for the formal verification of lemma `Step_Game1_Game2`.

```
Current goal

p : Rp
------------------------------------------------------------------------
scaleRp2R2t p = scaleRppq2R2t (scaleRp2Rppq p)
```

Listing 4.50: Example Goal 1

In this goal representation, the dotted line separates the goal's context (above the line) from the goal's conclusion (below the line). In accordance with the definition of the considered lemma, the goal's context declares, or "assumes", an artifact `p` of type `Rp` which, as such, may be employed in the goal's conclusion. Furthermore, since the context does not specify, or "assume", any additional restrictions on `p`, this artifact represents an arbitrary value of type `Rp`; alternatively stated, the goal's conclusion must hold for any value of type `Rp`. In fact, EasyCrypt allows for the transformation of this goal into an equivalent one that does not declare an artifact of type `Rp` in its context; instead, this goal's conclusion universally quantifies over `Rp`. Listing 4.51 presents this equivalent goal.

[11] Actually, the context of a goal additionally specifies a set of *type variables* that enable the context's assumptions to consider abstract types. Nevertheless, this feature has not been utilized throughout the entire formal verification effort for Saber.PKE; therefore, we disregard this technicality.

```
Current goal

------------------------------------------------------------------------
forall (p : Rp), scaleRp2R2 p = scaleRq2R2 (scaleRp2Rq p)
```

Listing 4.51: Example Goal 2, Equivalent to Example Goal 1

Indeed, this indicates that parameterizing a lemma is equivalent to universally quantifying over the type(s) of the parameter(s), as previously mentioned in Section 2.3.1.

pRHL Judgment Goals and pRHL Statement Judgment Goals

Throughout the demonstration of the formal verification of lemma `Step_Game1_Game2`, we will encounter several goals with a special representation. More concretely, this concerns goals with conclusions denoting either a *pRHL judgment* or a *pRHL statement judgment*, which EasyCrypt renders in a non-linear fashion; we respectively refer to such goals as "pRHL judgment goals" and "pRHL statement judgment goals". In essence, both types of judgment correspond to statements that are entirely defined by a single (use of the) `equiv` keyword; however, the difference between the two is that pRHL judgments consider the identifiers of procedures, while pRHL statement judgments consider the actual specifications of procedures[12]. Certainly, (the statements of) the equivalence lemmas discussed in Section 4.2.3 constitute pRHL judgments; as such, the initial goals in the proofs of these lemmas are subjected to the aforementioned non-linear rendering. Exemplifying this rendering, Listing 4.52 presents the initial goal in the proof of lemma `Equivalence_kg`; the statement corresponding to this lemma is `equiv[Saber_PKE_Scheme.kg ~ Saber_PKE_Scheme_Alt.kg : true ==> ={res}]`, as defined in Listing 4.36.

```
Current goal

------------------------------------------------------------------------
pre = true

    Saber_PKE_Scheme.kg ~ Saber_PKE_Scheme_Alt.kg

post = ={res}
```

Listing 4.52: Example pRHL Judgment Goal

Comparing the conclusion of the goal in this listing to the statement of the corresponding lemma, it is evident how EasyCrypt renders pRHL judgments as goal conclusions. Moreover, the non-linear rendering of these judgments accurately conveys their intuitive interpretation. Namely, an arbitrary (valid) pRHL judgment `equiv[M.a ~ N.b : phi ==> psi]` intuitively denotes that executing procedures `M.a` and `N.b` in any memories that satisfy predicate `phi` results in (sub-distributions on) memories that satisfy predicate `psi`; this suggests that `phi` and `psi` can be interpreted as a precondition and a postcondition, respectively[13]. Here, "memories that satisfy `phi`" refers to memories for which `phi` evaluates to true when all of the identifiers in this predicate are replaced by the corresponding concrete values from these memories; nevertheless, "(sub-distributions on) memories that satisfy `psi`" has, due to the consideration of sub-distributions on memories instead

[12]As indicated previously, these types of statements can be employed considerably more generally than necessary for the formal verification effort regarding Saber.PKE. In particular, these types of statements, and hence their generality, are formally and rigorously defined through an abstruse mathematical definition [18, 54]. Nevertheless, precluding gratuitous generalizations and expositions, we refrain from directly using or discussing this mathematical definition; instead, we adopt a more intuitive and concrete approach concerning these types of statements in this discussion, exclusively considering the relevant instances and applications.

[13]In the non-linear rendering of pRHL judgments as goal conclusions, these interpretations of `phi` and `psi` are accordingly expressed as `pre = phi` and `post = psi`.

of concrete memories, a rather intricate formal definition and does not allow for a straightforward intuitive interpretation. Fortunately, as can be extracted from Section 4.2, the preconditions and postconditions of the pRHL statements employed in the formal verification effort regarding Saber.PKE exclusively comprise (a conjunction of) equalities between values from the different memories. Consequently, in the context of this discussion, "(sub-distributions on) memories that satisfy `psi`" has the following relatively intuitive interpretation: For each equality in `psi`, the probability that the left-hand side is assigned a particular value by the corresponding memory is equal to the probability that the right-hand side is assigned this same value by the other memory.

Similarly considering lemma `Equivalence_kg` as an example, Listing 4.53 illustrates the non-linear rendering of a goal conclusion that corresponds to a pRHL statement judgment. In fact, the goal in this listing can be obtained by applying the `proc` tactic on the goal presented in Listing 4.52.

```
Current goal

------------------------------------------------------------------------
&1 (left ) : Saber_PKE_Scheme.kg
&2 (right) : Saber_PKE_Scheme_Alt.kg

pre = true

sd <$ dseed                        (1)  sd <$ dseed
_A <- gen sd                       (2)  _A <- gen sd
s <$ dsmallRq_vec                  (3)  s <$ dsmallRq_vec
b <- scaleRqv2Rpv (_A *^ s + h)    (4)  b <- scaleRqv2Rpv (_A *^  s +  h)

post =
  (pk_encode_s (sd{1}, b{1}), sk_encode_s s{1}) =
  (pk_encode_s (sd{2}, b{2}), sk_encode_s s{2})
```

Listing 4.53: Example pRHL Statement Judgment Goal

As we can see, (the rendering of) a pRHL statement judgment is considerably more thorough than (the rendering of) the corresponding pRHL judgment. First, as aforementioned, a pRHL statement judgment concerns the specifications of the considered procedures instead of their identifiers. Nevertheless, the rendering of such a judgment explicitly indicates both the identifiers and the specifications; in this case, the left specification corresponds to `Saber_PKE_Scheme.kg`, while the right specification corresponds to `Saber_PKE_Scheme_Alt.kg`. Second, although both the left and right procedures of a pRHL judgment (that is, left and right of the `~`) and a pRHL statement judgment invariably execute in memories denominated `&1` (left procedure) and `&2` (right procedure), only the rendering of a pRHL statement judgment explicitly states this fact. Notably, these memories are slightly different from the memories discussed hitherto. Namely, the memories considered thus far constitute, as initially described in Section 4.2, artifacts that assign values to all global variables declared by the modules specified in the currently considered script; these memories are utilized to, e.g., parameterize lemmas and specify certain probability statements. However, given such a general memory `&m`, the aforementioned memories `&1` and `&2` extend `&m` in the context of a procedure call by additionally assigning values to the considered procedure's parameters, local variables, and, if the procedure terminates, special artifact `res`; specifically, if the considered procedure terminates, `res` is assigned the procedure's return value. Nevertheless, concerning either type of memory, the concrete value of a certain artifact, e.g., `x`, within a particular memory, e.g., `&mem`, is denoted by `x{mem}`. In the remainder, we refer to the general, non-extended memories as "general memories" and to the procedure-specific, extended memories as "procedure-extended memories". Lastly, (the rendering of) a pRHL statement judgment replaces all references to `res` by the actual return value(s) of the corresponding procedures. As such, since `={res}` is equivalent to `res{1} = res{2}`, this becomes `(pk_encode_s (sd, b), sk_encode_s s){1} = (pk_` `encode_s (sd, b), sk_encode_s s){2}`; in turn, this latter equality is equivalent to the actual postcondition of the goal's conclusion in Listing 4.53.

Tactics

Albeit EasyCrypt implements an abundance of different tactics, the proof of lemma `Step_Game1_Game2` merely requires several of the more prevalent ones. In the ensuing, we concisely describe these tactics, primarily focalizing on the application of these tactics in the currently considered context, i.e., the demonstration of the proof of lemma `Step_Game1_Game2`. Particularly, certain tactics might be more generally applicable or support more features than indicated. Furthermore, to preclude extensive technical discussions, several tactic descriptions remain relatively intuitive.

For reasons of intelligibility, we divide the tactics into two different categories based on their applicability. More precisely, the first category contains tactics applicable to arbitrary goals; the second category comprises tactics exclusively applicable to certain probability-related goals, pRHL judgment goals, or pRHL statement judgment goals. Concerning the former, the following list covers the relevant tactics applicable to arbitrary goals.

- `trivial`
  This tactic solves a trivial goal by applying a combination of low-level tactics; these low-level tactics are not directly accessible by the user.

- `simplify`
  This tactic simplifies a goal's conclusion by reducing it to a canonical normal form; this exclusively affects the notation of the goal's conclusion, not its meaning.

  Demonstrating the utilization of this tactic, the following list constitutes several examples of expressions that are reduced to a simpler form when applying `simplify`.

  – **Example 1**
    If a goal's conclusion contains expressions of the form `if true then a else b`, applying `simplify` reduces these expressions to `a`; similarly, expressions of the form `if false then a else b` are reduced to `b`.

  – **Example 2**
    If a goal's conclusion comprises expressions of the form ``(a, b).`1``, applying `simplify` reduces these expressions to `a`; analogously, expressions of the form ``(a, b).`2`` are reduced to `b`.

- `rewrite`
  This tactic rewrites a goal's conclusion.

  Consider an identifier of an axiom, a lemma, or an assumption that refers to a statement `f = g`; moreover, consider a goal with a conclusion that contains `f` at least once. Then, applying `rewrite id` to this goal produces a single goal with the following context and conclusion.

  – **Goal**
    Context: Identical to the original goal's context.
    Conclusion: Identical to the original goal's conclusion, but with each occurrence of `f` replaced by `g`.

- `progress`
  This tactic breaks a goal into zero or more simplified goals by repeatedly applying several other tactics. Relative to the original goal's context, the contexts of the generated goals may

be unchanged, contain novel assumptions, have assumptions removed, or a combination of the latter two; similarly, compared to the original goal's conclusion, the conclusions of the generated goals may be unaltered or simplified.

- `have`
  This tactic introduces an additional assumption in a goal's context; as such, this tactic essentially constitutes a formalization of the logical cut inference rule.

  Consider a goal with conclusion `psi`; moreover, consider a predicate `phi` and an identifier `id` not yet in the considered goal's context. Then, applying `have id: phi` to this goal produces two goals with the following contexts and conclusions.

  - **Goal 1**
    Context: Identical to the original goal's context.
    Conclusion: `phi`

  - **Goal 2**
    Context: Identical to the original goal's context extended with the assumption `id: phi`.
    Conclusion: `psi`

- `congr`
  This tactic removes the application of an operator in a goal's conclusion; this is based on the fact that, in EasyCrypt, operators are mathematical functions.

  Consider a goal with conclusion `f x_1 ... x_n = f y_1 ... y_n` or `f x_1 ... x_n <=> f y_1 ... y_n`, where `f` is an operator and $1 \leq$ `n`. Then, applying `congr` to this goal produces `n` goals with the following context(s) and conclusion(s).

  - **Goal** `i` ($1 \leq$ `i` $\leq$ `n`)
    Context: Identical to the original goal's context.
    Conclusion: `x_i = y_i`.

Regarding the remaining category, the ensuing list describes the relevant tactics exclusively applicable to particular probability-related goals, pRHL judgment goals or pRHL statement judgment goals.

- `byequiv`
  This tactic transforms a probability-related goal into a pRHL judgment goal.

  Consider a general memory `&m`, predicates `phi` and `psi`, and modules `M` and `N` with, respectively, procedures `a(x_1, ..., x_n)` and `b(x_1, ..., x_n)`; for these procedures, the parameter list might be empty, i.e., `n` might be equal to 0. Furthermore, consider a goal with conclusion `Pr[M.a(x_1, ..., x_n) @ &m : res] = Pr[N.b(x_1, ..., x_n) @ &m : res]`. Then, applying `byequiv (_ : phi ==> psi)` to this goal produces three goals with the following contexts and conclusions.

  - **Goal 1**
    Context: Identical to the original goal's context.
    Conclusion: A pRHL judgment `equiv[M.a ~ N.b : phi ==> psi]`.

  - **Goal 2**
    Context: Identical to the original goal's context.
    Conclusion: `phi`.

  - **Goal 3**
    Context: Identical to the original goal's context.
    Conclusion: For all procedure-extended memories `&1` (for `M.a`) and `&2` (for `N.b`), `psi ==> ={res}`.

  Alternatively, applying `byequiv`, i.e., omitting `phi` and `psi`, enforces EasyCrypt (to attempt to) infer `phi` and `psi`. In this case, `phi` typically denotes a relation between the (actual) parameters; moreover, oftentimes, `psi` straightforwardly equals `={res}`.

- `proc`
  This tactic replaces a pRHL judgment goal by an equivalent pRHL statement judgment goal.

  Consider a pRHL judgment goal `equiv[M.a ~ N.b : phi ==> psi]` where the identifiers denote the same artifacts as in the explanation of `byequiv` above. Then, applying `proc` to this goal produces a single goal with the following context and conclusion.

  - **Goal**
    Context: Identical to the original goal's context.
    Conclusion: A pRHL statement judgment that arises from, in the original goal's conclusion, replacing `M.a` and `N.b` by their specification.

- `inline`
  In a pRHL statement judgment goal, this tactic replaces a call to a (concrete) procedure with the specification of this procedure; naturally, each of the specification's formal parameters is substituted with the corresponding argument provided in the call.

  Consider a pRHL statement judgment goal with procedure specifications containing a (concrete) procedure call `M.a(x_1, ..., x_n)`. Then, applying `inline M.a` to this goal produces one goal with the following context and conclusion.

  - **Goal**
    Context: Identical to the original goal's context.
    Conclusion: A pRHL statement judgment that arises from, in the original goal's conclusion, replacing `M.a` by its specification; moreover, in this specification, each formal parameter is substituted with the corresponding argument provided in the original procedure call.

- `wp`
  This tactic removes the longest suffixes of regular assignment statements and if-else control structures from the procedure specifications considered in a pRHL statement judgment goal; afterward, the postcondition is appropriately adjusted. Specifically, the postcondition of the resulting pRHL statement judgment goal is the *weakest precondition* required for the original postcondition to hold if the removed suffixes were to be executed (starting from this weakest precondition); here, "weakest" intuitively refers to "least restrictive".

- `auto`
  This tactic simplifies a pRHL statement judgment goal by repeatedly applying numerous different tactics; an example of such a different tactic is the above-mentioned `wp`. More precisely, application of this tactic removes the longest suffixes without call statements from the considered procedure specifications; subsequently, akin to `wp`, it sets the postcondition to the weakest precondition for the original postcondition to hold if the removed suffixes were to be executed (commencing from this weakest precondition). Moreover, in case both of the considered procedure specifications are empty, either preceding or during the application of `auto`, this tactic generates an equivalent general goal; that is, if the precondition and

postcondition of such a pRHL judgment statement goal respectively equal `phi` and `psi`, `auto` produces a goal with conclusion `phi ==> psi`[14].

- `call`
  This tactic removes the last statement of each of the considered procedure specifications in a pRHL statement judgment goal, provided these statements constitute calls to the same abstract procedure. Specifically, for a predicate `phi`, application of `call (_ : phi)` to such a pRHL statement judgment goal initially removes the last statement of each of the considered procedure specifications. Thereafter, the postcondition is adjusted to essentially require that if the (removed) procedures were to be called, `phi` would hold beforehand and afterward, the procedures would behave identically, and the return values of the procedures would be equal; additionally, the resulting postcondition necessitates that these properties imply the original postcondition. In order to guarantee that two calls to the same abstract procedure in different (procedure-extended) memories behave identically, both the arguments provided in the calls and the accessible global variables of the corresponding module must be equal between these memories. Here, remark the similarities with the discussion presented in Section 3.2.2, particularly concerning the segment on the requirements for an adversary to behave identically between a reduction and a run of its own game; indeed, "the arguments provided in the (abstract procedure) calls and the accessible global variables of the corresponding module must be equal between the considered (procedure-extended) memories" is EasyCrypt's way of generically formalizing "the information provided/available to an adversary must be indistinguishable (to this adversary) between a reduction and a run of its own game".

4.3.2 Proof of `Step_Game1_Game2` Lemma

Leveraging the concepts and mechanisms discussed in the preceding section, we demonstrate the concrete formal verification of lemma `Step_Game1_Game2` in EasyCrypt.

First, we reiterate the specification of `Step_Game1_Game2` in Listing 4.54; however, in contrast to the original specification of this lemma in Listing 4.30, this reiteration includes the concrete proof of the lemma. Furthermore, for clarity purposes, this proof is commented and indented, indicating its structure and flow.

```
lemma Step_Game1_Game2 &m (A <: Adversary) :
      `| Pr[Game1(A).main() @ &m : res] - 1%r / 2%r |
      =
      `| Pr[Game2( A2(A) ).main() @ &m : res] - 1%r / 2%r |.
proof.
(* Introduce useful assumption into goal's context with have *)
have eq_pr: Pr[Game1(A).main() @ &m : res] = Pr[Game2( A2(A) ).main() @ &m : res].

  (* Proof of assumption introduced by have *)
  byequiv; trivial.
  proc.
  inline A2(A).choose; inline A2(A).guess.
  wp.
  call (_ : true).
  auto.
  call (_ : true).
  auto.
  progress.
```

[14]Albeit `auto` can automatically process a considerable portion of most pRHL statement judgment goals, it does so in a generic manner; oftentimes, to reach the desired conclusions, the processing of these goals requires more granular approaches. As such, although a powerful tactic, it is not a panacea for solving pRHL statement judgment goals.

```
  congr; congr.

    (* Proof of equality first (output) tuple element *)
    rewrite cg_enc_dec_inv; simplify.
    rewrite scaleRp2Rppq2R2t_comp.
    trivial.

    (* Proof of equality second (output) tuple element *)
    rewrite cg_enc_dec_inv.
    trivial.

(* Proof of original claim using assumption introduced by initial have *)
rewrite eq_pr.
trivial.
qed.
```

Listing 4.54: Proof Demonstration 0 – Specification of `Step_Game1_Game2` Including Proof

Commencing the formal verification process regarding this lemma, EasyCrypt generates the initial goal in accordance with the lemma's specification. Namely, as depicted in Listing 4.55, the generated goal's context declares a memory `&m` and an `Adversary` module `A`, indicating the universal quantification over both memories and modules of type `Adversary`; indeed, this correctly corresponds to the parameterization of the lemma. Furthermore, the generated goal's conclusion precisely equals the lemma's statement.

```
Current goal

&m: memory
A : Adversary
------------------------------------------------------------------------
`|Pr[Game1(A).main() @ &m : res] - 1%r / 2%r| =
`|Pr[Game2(A2(A)).main() @ &m : res] - 1%r / 2%r|
```

Listing 4.55: Proof Demonstration 1 – Initial Goal

Examining the conclusion of this goal, we see that it becomes trivial if the two probability statements are identical. As such, utilizing the `have` tactic, the first step of the proof concerns introducing the assumption `Pr[Game1(A).main() @ &m : res] = Pr[Game2(A2(A)).main() @ &m : res]` into the goal's context. As can be extracted from Listing 4.54, we befittingly denominate this assumption `eq_pr`. Applying this instance of the `have` tactic, i.e., `have eq_pr: Pr[Game1(A).main() @ &m : res] = Pr[Game2( A2(A) ).main() @ &m : res]`, generates the goals presented in Listing 4.56.

```
Current goal (remaining: 2)

&m: memory
A : Adversary
------------------------------------------------------------------------
Pr[Game1(A).main() @ &m : res] = Pr[Game2(A2(A)).main() @ &m : res]



Goal #2
------------------------------------------------------------------------
`|Pr[Game1(A).main() @ &m : res] - 1%r / 2%r| =
`|Pr[Game2(A2(A)).main() @ &m : res] - 1%r / 2%r|
```

Listing 4.56: Proof Demonstration 2 – Goals After `have` Tactic

As this listing suggests, in case multiple goals remain to be verified, EasyCrypt merely renders the context of a single goal, the “primary goal”[15]; for the remaining goals, only the conclusion is rendered. Furthermore, the application of a tactic exclusively affects the primary goal; at least, this applies to the first tactic in a chain. Namely, EasyCrypt allows for the chaining of tactics by separating them with a semicolon. Essentially, each tactic following a `;` is applied to every goal generated in that chain of tactics. The ensuing step in the proof exemplifies this mechanism. In particular, as shown in Listing 4.56, the current primary goal constitutes an equality of probability statements; as such, we can progress the proof by employing the `byequiv` tactic, transforming this goal into a pRHL judgment goal. Furthermore, in this case, EasyCrypt's inference mechanism for the precondition and postcondition of this pRHL judgment goal is apt for the formal verification of this lemma; that is, it suffices to utilize the `byequiv` as is, without manually specifying the precondition and postcondition. Nevertheless, the application of this tactic produces three goals, two of which are trivial. Consequently, chaining the `byequiv` tactic with the `trivial` tactic immediately solves two of the three generated goals. Importantly, if trivial is applied to a tactic that it cannot solve, it does not fail or throw an error; instead, it simply returns and leaves the goal unaltered. For this reason, even though one of the three goals generated by `byequiv` is not solvable through `trivial`, we can still safely chain these tactics. Applying the `byequiv; trivial` chain results in the set of remaining goals provided in Listing 4.57.

```
Current goal (remaining: 2)

&m: memory
A : Adversary
------------------------------------------------------------------------
pre = (glob A){2} = (glob A){m} /\ (glob A){1} = (glob A){m}

    Game1(A).main ~ Game2(A2(A)).main

post = ={res}



Goal #2
------------------------------------------------------------------------
`|Pr[Game1(A).main() @ &m : res] - 1%r / 2%r| =
`|Pr[Game2(A2(A)).main() @ &m : res] - 1%r / 2%r|
```

Listing 4.57: Proof Demonstration 3 – Goals After Chain of `byequiv` and `trivial` Tactics

As we can see, the application exclusively affected the primary goal; moreover, the chaining of `trivial` indeed solved two out of three generated goals. Hereafter, for reasons of conciseness, we will omit any non-primary goals that have already been shown in a previous listing. Considering the primary goal in Listing 4.57, we notice an unprecedented artifact, viz., `glob A`. This artifact denotes the aforementioned set of global variables that module `A` can access, including any global variables declared by other modules[16]. Moreover, for a memory `&mem`, `(glob A){mem}` signifies this same set, yet with each variable assigned a concrete value in accordance with `mem`. As such, the precondition of the pRHL judgment in the above goal's conclusion essentially states that the set of global variables accessible by `A` is identical between the two considered procedure-extended memories; certainly, from both of these procedure-extended memories, the variables in this set are assigned the same concrete values as from the general memory `&m`. Conceptually, as alluded to before, this suggests that the adversary against the first game operates in the same (global) context for both games.

[15]This is by no means a conventional term; however, we introduce it for convenience purposes.

[16]More precisely, this set includes any global variables declared by other modules for which access is not explicitly denied; albeit not relevant in this case, an example of such explicit denial is given in Listing 4.26.

In order to proceed, we require the actual specification of the considered procedures rather than their identifiers; that is, we desire to transform the pRHL judgment goal into a corresponding pRHL statement judgment goal. As such, we apply the `proc` tactic; moreover, to obtain the full specifications, we subsequently inline the concrete procedure calls, i.e., `A2(A).choose(pk_encode_g (sd, b))` and `A2(A).guess(c_encode_g (cmu, b'))`, through chaining two appropriate `inline` tactics. The goal generated by this sequence of tactic applications is provided in Listing 4.58.

```
Current goal (remaining: 2)

&m: memory
A : Adversary
------------------------------------------------------------------------
&1 (left ) : Game1(A).main
&2 (right) : Game2(A2(A)).main

pre = (glob A){2} = (glob A){m} /\ (glob A){1} = (glob A){m}

u <$ {0,1}                          ( 1) u <$ {0,1}
sd <$ dseed                         ( 2) sd <$ dseed
_A <- gen sd                        ( 3) _A <- gen sd
b <$ dRp_vec                        ( 4) b <$ dRp_vec
(m0, m1) <@ A.choose(pk_encode_g    ( 5) pk <- pk_encode_g (sd, b)
             (sd, b))               (  )
s' <$ dsmallRq_vec                  ( 6) (m00, m10) <@ A.choose(pk)
b' <- scaleRqv2Rpv (trmx _A *^ s'   ( 7) (m0, m1) <- (m00, m10)
       + h)                         (  )
v' <- dotp b (Rqv2Rpv s')           ( 8) s' <$ dsmallRq_vec
       + Rq2Rp h1                   (  )
cmu <- scaleRp2R2t (v'              ( 9) b' <- scaleRqv2Rpv (trmx
        + scaleR22Rp (m_decode      (  )        _A *^ s' + h)
        (if u then m1 else m0)))    (  )
u' <@ A.guess(c_encode_g (cmu, b')) (10) v' <- dotp b (Rqv2Rpv s')
                                    (  )        + Rq2Rp h1
                                    (11) cmu <- scaleRp2Rppq (v'
                                    (  )         + scaleR22Rp (m_decode
                                    (  )         (if u then m1 else m0)))
                                    (12)  c <- c_encode_g (cmu, b')
                                    (13)  c_dec <- c_decode_g c
                                    (14)  cmu0 <- c_dec.`1
                                    (15)  b'0 <- c_dec.`2
                                    (16)  cmu' <- scaleRppq2R2t cmu0
                                    (17)  u'0 <@ A.guess(c_encode_g(
                                    (  )          cmu', b'0))
                                    (18)  u' <- u'0

post = (u{1} = u'{1}) = (u{2} = u'{2})
```

Listing 4.58: Proof Demonstration 4 – Goals After `proc` and (Chain of) `inline` Tactics

As desired, comparing the left and right program specifications of the above pRHL statement judgment goal to the specifications of `Game1(A).main` and, respectively, `Game2(A2(A)).main` (after inlining the procedures of `A2(A)`), we see that they are indeed identical; moreover, both the left-hand and right-hand sides of the postcondition's equality in Listing 4.57, i.e., `res{1} = res{2}`, are accordingly replaced in Listing 4.58, viz., they are replaced by the respective return values of the corresponding procedures.

Inspecting the procedure specifications of the pRHL statement judgment goal in Listing 4.58, we observe that the left specification ends with the abstract procedure call `A.guess(c_encode_g (cmu, b'))`, while the right specification ends with the regular assignment `u' <- u'0`. Nevertheless, to show that this abstract procedure call behaves identically to the analogous procedure call in the right specification, i.e., `A.guess(c_encode_g(cmu', b'0))`, we must remove them simultaneously by applying the `call` tactic. Since this application requires both calls to be the final statement of the considered specifications, we foremost desire to remove the `u' <- u'0` from the right specification; this can be accomplished through the `wp` tactic. Listing 4.59 shows the result of applying this tactic.

```
Current goal (remaining: 2)

&m: memory
A : Adversary
------------------------------------------------------------------------
&1 (left ) : Game1(A).main
&2 (right) : Game2(A2(A)).main

pre = (glob A){2} = (glob A){m} /\ (glob A){1} = (glob A){m}

u <$ {0,1}                          ( 1) u <$ {0,1}
sd <$ dseed                         ( 2) sd <$ dseed
_A <- gen sd                        ( 3) _A <- gen sd
b <$ dRp_vec                        ( 4) b <$ dRp_vec
(m0, m1) <@ A.choose(pk_encode_g    ( 5) pk <- pk_encode_g (sd, b)
             (sd, b))               (  )
s' <$ dsmallRq_vec                  ( 6) (m00, m10) <@ A.choose(pk)
b' <- scaleRqv2Rpv (trmx _A *^ s'   ( 7) (m0, m1) <- (m00, m10)
       + h)                         (  )
v' <- dotp b (Rqv2Rpv s')           ( 8) s' <$ dsmallRq_vec
       + Rq2Rp h1                   (  )
cmu <- scaleRp2R2t (v'              ( 9) b' <- scaleRqv2Rpv (trmx
        + scaleR22Rp (m_decode      (  )        _A *^ s' + h)
        (if u then m1 else m0)))    (  )
u' <@ A.guess(c_encode_g (cmu, b')) (10) v' <- dotp b (Rqv2Rpv s')
                                    (  )         + Rq2Rp h1
                                    (11) cmu <- scaleRp2Rppq (v'
                                    (  )          + scaleR22Rp (m_decode
                                    (  )          (if u then m1 else m0)))
                                    (12)  c <- c_encode_g (cmu, b')
                                    (13)  c_dec <- c_decode_g c
                                    (14)  cmu0 <- c_dec.`1
                                    (15)  b'0 <- c_dec.`2
                                    (16)  cmu' <- scaleRppq2R2t cmu0
                                    (17)  u'0 <@ A.guess(c_encode_g(
                                    (  )           cmu', b'0))

post = (u{1} = u'{1}) = (u{2} = u'0{2})
```

Listing 4.59: Proof Demonstration 5 – Goals After `wp` Tactic

From this listing, it is apparent that the application of the `wp` tactic had the anticipated effect. Namely, this application removed the longest suffixes consisting of regular assignments and if-else control structures from both procedure specifications; in this case, this merely amounted to the removal of the aforementioned `u' <- u'0` assignment. Furthermore, with the replacement of `u'` by `u'0`, the postcondition was altered accordingly; that is, assuming a scenario in which this postcondition holds, executing the removed assignment would validate the original postcondition.

Succeeding the removal of `u' <- u'0`, both procedure specifications conclude with a call to the abstract procedure `A.guess`, albeit with different (identifiers of) variables as arguments. As alluded to before, we aim to ascertain that these procedures calls behave identically, formalizing the guaranteed identical behavior of the corresponding adversarial algorithm between the reduction and a run of its own game. To this end, we employ the `call` tactic; particularly, we apply `call (_ : true)`. Here, the provided predicate constitutes the trivial `true` predicate because the formal verification does not require any properties to be maintained by the procedure calls; that is, the formal verification solely necessitates the identical behavior of these calls. The goal generated by applying the `call (_ : true)` tactic is given in Listing 4.60.

```
Current goal (remaining: 2)

&m: memory
A : Adversary
------------------------------------------------------------------------
&1 (left ) : Game1(A).main
&2 (right) : Game2(A2(A)).main

pre = (glob A){2} = (glob A){m} /\ (glob A){1} = (glob A){m}

u <$ {0,1}                          ( 1) u <$ {0,1}
sd <$ dseed                         ( 2) sd <$ dseed
_A <- gen sd                        ( 3) _A <- gen sd
b <$ dRp_vec                        ( 4) b <$ dRp_vec
(m0, m1) <@ A.choose(pk_encode_g    ( 5) pk <- pk_encode_g (sd, b)
             (sd, b))               (  )
s' <$ dsmallRq_vec                  ( 6) (m00, m10) <@ A.choose(pk)
b' <- scaleRqv2Rpv (trmx _A *^ s'   ( 7) (m0, m1) <- (m00, m10)
       + h)                         (  )
v' <- dotp b (Rqv2Rpv s')           ( 8) s' <$ dsmallRq_vec
       + Rq2Rp h1                   (  )
cmu <- scaleRp2R2t (v'              ( 9) b' <- scaleRqv2Rpv (trmx
        + scaleR22Rp (m_decode      (  )        _A *^ s' + h)
        (if u then m1 else m0)))    (  )
                                    (10) v' <- dotp b (Rqv2Rpv s')
                                    (  )         + Rq2Rp h1
                                    (11) cmu <- scaleRp2Rppq (v'
                                    (  )          + scaleR22Rp (m_decode
                                    (  )          (if u then m1 else m0)))
                                    (12)  c <- c_encode_g (cmu, b')
                                    (13)  c_dec <- c_decode_g c
                                    (14)  cmu0 <- c_dec.`1
                                    (15)  b'0 <- c_dec.`2
                                    (16)  cmu' <- scaleRppq2R2t cmu0

post =
  (c_encode_g (cmu{1}, b'{1}) = c_encode_g (cmu'{2}, b'0{2}) /\ ={glob A} /\ true)
  &&
  forall (result_L result_R : bool) (A_L A_R : (glob A)),
    result_L = result_R /\ A_L = A_R /\ true => (u{1} = result_L) = (u{2} = result_R)
```

Listing 4.60: Proof Demonstration 6 – Goals After `call` Tactic

As can be extracted from this listing, the abstract procedure calls are removed from the procedure specifications and the postcondition is appropriately adjusted. In particular, consistent with the description of the `call` tactic, the generated postcondition states that the execution of the current procedures must result in the equality of the arguments originally provided to the (removed) abstract procedure calls, i.e., `c_encode_g (cmu{1}, b'{1})` and `c_encode_g (cmu'{2}, b'0{2})`; moreover, the accessible global variables of the module encompassing the abstract procedure, i.e., `glob A`, must have identical values between the two considered procedure-extended memories. Indeed, these requirements guarantee that the abstract procedure calls would behave the same, and hence produce identical results, if they were to be executed. Additionally, the postcondition requires that this identical behavior, irrespective of the concrete value that would have been produced by the (removed) abstract procedure calls, implies the postcondition of (the conclusion of) the goal preceding the application of `call`.

Considering the procedure specifications in Listing 4.60, we recognize that the suffixes of both specifications constitute a sequence of ordinary assignments and random samplings. As such, the `auto` tactic can be applied to discard these suffixes and appropriately change the postcondition; the precise effect of this application is exhibited in Listing 4.61.

```
Current goal (remaining: 2)

&m: memory
A : Adversary
------------------------------------------------------------------------
&1 (left ) : Game1(A).main
&2 (right) : Game2(A2(A)).main

pre = (glob A){2} = (glob A){m} /\ (glob A){1} = (glob A){m}

u <$ {0,1}                            (1) u <$ {0,1}
sd <$ dseed                           (2) sd <$ dseed
_A <- gen sd                          (3) _A <- gen sd
b <$ dRp_vec                          (4) b <$ dRp_vec
(m0, m1) <@ A.choose(pk_encode_g      (5) pk <- pk_encode_g (sd, b)
             (sd, b))                 ( )
                                      (6) (m00, m10) <@ A.choose(pk)

post =
  (forall (s'R : Mat_Rq.vector), s'R \in dsmallRq_vec => s'R = s'R) &&
  (forall (s'R : Mat_Rq.vector),
     s'R \in dsmallRq_vec => mu1 dsmallRq_vec s'R = mu1 dsmallRq_vec s'R) &&
  forall (s'L : Mat_Rq.vector),
    s'L \in dsmallRq_vec => (s'L \in dsmallRq_vec) && s'L = s'L &&
    let c_dec_R = c_decode_g (c_encode_g (scaleRp2Rppq (dotp b{2} (Rqv2Rpv s'L)
    + Rq2Rp h1 + scaleR22Rp (m_decode (if u{2} then m10{2} else m00{2}))),
    scaleRqv2Rpv (trmx _A{2} *^ s'L + h))) in
    (c_encode_g (scaleRp2R2t (dotp b{1} (Rqv2Rpv s'L) + Rq2Rp h1 +
     scaleR22Rp (m_decode (if u{1} then m1{1} else m0{1}))),
     scaleRqv2Rpv (trmx _A{1} *^ s'L + h))
     =
     c_encode_g (scaleRppq2R2t c_dec_R.`1, c_dec_R.`2) /\ ={glob A}) &&
     forall (result_L result_R : bool) (A_L A_R : (glob A)),
      result_L = result_R /\ A_L = A_R => (u{1} = result_L) = (u{2} = result_R)
```

Listing 4.61: Proof Demonstration 7 – Goals After `auto` Tactic

As expected, the application of the `auto` tactic removed the suffixes of both procedures specifications up until the calls to `A.choose`, which this tactic cannot process; in addition, the postcondition was changed accordingly. Evidently, the resulting postcondition is quite complex and illegible; this is predominantly due to the relative extensiveness of the removed suffixes. Nevertheless, the procedure by which this postcondition was obtained is rather straightforward. Namely, in essence, this postcondition arises from sequentially replacing the artifacts in the preceding postcondition (i.e., the postcondition considered in Listing 4.60) by the values or expressions assigned to them in the statements of the removed suffixes. Moreover, for the removed random samplings of `s'`, the postcondition in Listing 4.61 contains a requirement stating that these samplings were equivalent between the two procedures; since these samplings were identical, this trivially holds. Although we could endeavor to manually simplify this postcondition to get a better understanding of its interpretation, the `progress` tactic will automatically perform this simplification momentarily.

At this point, following the same line of reasoning as before, we apply the `call (_ : true)` to simultaneously remove the abstract procedures calls `A.choose(pk_encode_g (sd, b))` and `A.choose(pk)`. Afterward, the remainder of the considered procedure specifications exclusively consists of regular assignments and random samplings. As such, directly following to the application of `call (_ : true)`, we apply the `auto` tactic anew. Listing 4.62 depicts the goal resulting from this sequence of applications.

```
Current goal (remaining: 2)

&m: memory
A : Adversary
------------------------------------------------------------------------
forall &1 &2,
  (glob A){2} = (glob A){m} /\ (glob A){1} = (glob A){m} =>
  (forall (uR : bool), uR \in {0,1} => uR = uR) &&
  forall (uL : bool),
  uL \in {0,1} => uL = uL &&
  (forall (sdR : seed), sdR \in dseed => sdR = sdR) &&
  forall (sdL : seed), sdL \in dseed => sdL = sdL &&
  let _A_R = gen sdL in (forall (bR : vector),
  bR \in dRp_vec => bR = bR) && (forall (bR : vector), bR \in dRp_vec =>
  mu1 dRp_vec bR = mu1 dRp_vec bR) && forall (bL : vector),
  bL \in dRp_vec => (bL \in dRp_vec) && bL = bL &&
  (pk_encode_g (sdL, bL) = pk_encode_g (sdL, bL) /\ ={glob A}) &&
  forall (result_L result_R : plaintext * plaintext) (A_L A_R : (glob A)),
  result_L = result_R /\ A_L = A_R => forall (s'L : Mat_Rq.vector),
  s'L \in dsmallRq_vec => (s'L \in dsmallRq_vec) && (c_encode_g
  (scaleRp2R2t (dotp bL (Rqv2Rpv s'L) + Rq2Rp h1 + scaleR22Rp
  (m_decode (if uL then result_L.`2 else result_L.`1))),
  scaleRqv2Rpv (trmx (gen sdL) *^ s'L + h)) = c_encode_g
  (scaleRppq2R2t (c_decode_g (c_encode_g (scaleRp2Rppq (dotp bL
  (Rqv2Rpv s'L) + Rq2Rp h1 + scaleR22Rp (m_decode
  (if uL then result_R.`2 else result_R.`1))),  scaleRqv2Rpv
  (trmx _A_R *^ s'L + h)))).`1, (c_decode_g (c_encode_g (scaleRp2Rppq
  (dotp bL (Rqv2Rpv s'L) + Rq2Rp h1 + scaleR22Rp (m_decode
  (if uL then result_R.`2 else result_R.`1))), scaleRqv2Rpv
  (trmx _A_R *^ s'L + h)))).`2) /\ A_L = A_R) &&
  forall (result_L0 result_R0 : bool) (A_L0 A_R0 : (glob A)),
  result_L0 = result_R0 /\ A_L0 = A_R0 => (uL = result_L0) = (uL = result_R0)
```

Listing 4.62: Proof Demonstration 8 – Goals After `call` and `auto` Tactics

Notably, the goal in above listing is not a pRHL statement judgment goal; instead, it is a general goal, i.e., its conclusion is an ordinary predicate. This is a consequence of the application of `auto`. Namely, as mentioned before, this tactic is applied after the removal of the calls to `A.choose`, i.e., when the considered procedure specifications exclusively constitute regular assignments and random samplings. Since the `auto` tactic is capable of processing and removing all of these assignments and samplings, its application reaches a point at which both procedure specification are empty. At such a point, as discussed in the description of `auto`, the application of this tactic constructs a general goal from the considered pRHL statement judgment goal; more precisely, assuming the precondition and postcondition of the considered pRHL statement judgment goal respectively equal `phi` and `psi`, the application of `auto` constructs a general goal with conclusion `phi ==> psi`. Examining the conclusion of the goal provided in Listing 4.62, we recognize that it indeed adheres to this format; namely, this conclusion is constructed as `forall &1 &2, (glob A){2} = (glob A){m} /\ (glob A){1} = (glob A){m} ==> (...)`, where the predicate preceding the implication accurately matches the precondition of the pRHL statement judgment goals from the preceding listings[17].

Albeit the goal given in Listing 4.62 is vastly convoluted and (nearly) indecipherable, the application of the `progress` tactic significantly extends the goal's context and considerably simplifies the goal's conclusion. Specifically, this application generates the goal presented in Listing 4.63.

```
Current goal (remaining: 2)

&m: memory
A : Adversary
&1: memory <Game1(A).main>
&2: memory <Game2(A2(A)).main>
uL: bool
H : uL \in {0,1}
sdL: seed
H0: sdL \in dseed
bL: vector
H1: bL \in dRp_vec
H2: bL \in dRp_vec
result_R: plaintext * plaintext
A_R: (glob A)
s'L: Mat_Rq.vector
H3: s'L \in dsmallRq_vec
H4: s'L \in dsmallRq_vec
------------------------------------------------------------------------
c_encode_g
  (scaleRp2R2t (dotp bL (Rqv2Rpv s'L) + Rq2Rp h1
   + scaleR22Rp (m_decode (if uL then result_R.`2 else result_R.`1))),
  scaleRqv2Rpv (trmx (gen sdL) *^ s'L + h))
=
c_encode_g
  (scaleRppq2R2t (c_decode_g (c_encode_g
   (scaleRp2Rppq (dotp bL (Rqv2Rpv s'L) + Rq2Rp h1
    + scaleR22Rp (m_decode (if uL then result_R.`2 else result_R.`1))),
  scaleRqv2Rpv (trmx (gen sdL) *^ s'L + h)))).`1,
  (c_decode_g (c_encode_g (scaleRp2Rppq (dotp bL (Rqv2Rpv s'L) + Rq2Rp h1
   + scaleR22Rp (m_decode (if uL then result_R.`2 else result_R.`1))),
  scaleRqv2Rpv (trmx (gen sdL) *^ s'L + h)))).`2)
```

Listing 4.63: Proof Demonstration 9 – Goals After `progress` Tactic

Collating Listing 4.62 with Listing 4.63, we see that the assumptions introduced into the context by the application of `progress` primarily originate from the explicit universal quantifications in the conclusion of the previous goal. The remainder of the introduced assumptions indicate that certain artifacts are elements of the set over which a particular distribution is defined; indeed, these assumptions correspond to the similarly constructed expressions in the conclusion of the preceding goal, i.e., the expressions utilizing `\in`. Furthermore, concerning the conclusion of the

[17] Remark that, although implicitly, pRHL statement judgment goals do universally quantify over the procedure-extended memories; consequently, the `forall &1 &2` in the general goal's conclusion is warranted.

goal in Listing 4.63, the improvement on the intelligibility relative to the conclusion of the goal in Listing 4.62 is evident.

Scrutinizing the goal in Listing 4.63, we observe that its conclusion has the form `c_encode_g (...) = c_encode_g (...)`; consequently, we can progress by applying the `congr` tactic, generating a goal of which the conclusion equates the considered arguments. Subsequently, because these arguments constitute 2-tuples, we can apply the `congr` tactic anew to generate two goals of which the conclusions equate the respective elements of the tuples. This consecutive application of the `congr` tactic can be performed at once by chaining two instances of this tactic; the ensuing listing presents the goals induced by the application of this tactic chain.

```
Current goal (remaining: 3)

&m: memory
A : Adversary
&1: memory <Game1(A).main>
&2: memory <Game2(A2(A)).main>
uL: bool
H : uL \in {0,1}
sdL: seed
H0: sdL \in dseed
bL: vector
H1: bL \in dRp_vec
H2: bL \in dRp_vec
result_R: plaintext * plaintext
A_R: (glob A)
s'L: Mat_Rq.vector
H3: s'L \in dsmallRq_vec
H4: s'L \in dsmallRq_vec
------------------------------------------------------------------------
scaleRp2R2t (dotp bL (Rqv2Rpv s'L) + Rq2Rp h1
  + scaleR22Rp (m_decode (if uL then result_R.`2 else result_R.`1)))
=
scaleRppq2R2t (c_decode_g (c_encode_g
  (scaleRp2Rppq (dotp bL (Rqv2Rpv s'L) + Rq2Rp h1
  + scaleR22Rp (m_decode (if uL then result_R.`2 else result_R.`1))),
  scaleRqv2Rpv (trmx (gen sdL) *^ s'L + h)))).`1



Goal #2
------------------------------------------------------------------------
scaleRqv2Rpv (trmx (gen sdL) *^ s'L + h)
=
(c_decode_g (c_encode_g (scaleRp2Rppq (dotp bL (Rqv2Rpv s'L) + Rq2Rp h1
  + scaleR22Rp (m_decode (if uL then result_R.`2 else result_R.`1))),
  scaleRqv2Rpv (trmx (gen sdL) *^ s'L + h)))).`2
```

Listing 4.64: Proof Demonstration 10 – Goals After Chain of `congr` Tactics

Regarding the primary goal generated by the chain of `congr`, the right-hand side of the conclusion's equality comprises the successive application of `c_decode_g` and `c_encode_g`. Axiomatically, these operators are each other's inverse and, as such, their consecutive application reduces to the identity function. Analogous to the axioms specified in Listing 4.10, `axiom cg_enc_dec_inv ['a] : cancel c_encode_g<:'a> c_decode_g<:'a>` constitutes the axiom stating this inverse property. Per the definition of `cancel` defined in EasyCrypt's standard library, `cg_enc_dec_inv` effectively denotes that for any `x` (of any type), `c_decode_g (c_encode_g x) = x`; thus, this axiom is compatible with the `rewrite` tactic. In fact, applying a chain of `rewrite cg_enc_dec_inv` and `simplify` to the primary goal in Listing 4.64 produces the goal depicted in Listing 4.65. Here, we additionally apply `simplify` to obtain a more convenient representation of the generated goal's conclusion.

```
Current goal (remaining: 3)

&m: memory
A : Adversary
&1: memory <Game1(A).main>
&2: memory <Game2(A2(A)).main>
uL: bool
H : uL \in {0,1}
sdL: seed
H0: sdL \in dseed
bL: vector
H1: bL \in dRp_vec
H2: bL \in dRp_vec
result_R: plaintext * plaintext
A_R: (glob A)
s'L: Mat_Rq.vector
H3: s'L \in dsmallRq_vec
H4: s'L \in dsmallRq_vec
------------------------------------------------------------------------
scaleRp2R2t (dotp bL (Rqv2Rpv s'L) + Rq2Rp h1
  + scaleR22Rp (m_decode (if uL then result_R.`2 else result_R.`1)))
=
scaleRppq2R2t (scaleRp2Rppq (dotp bL (Rqv2Rpv s'L) + Rq2Rp h1
  + scaleR22Rp (m_decode (if uL then result_R.`2 else result_R.`1))))
```

Listing 4.65: Proof Demonstration 11 – Goals After Chain of `rewrite` and `simplify` Tactics

At this point, the sole difference between the left- and right-hand sides of the conclusion's equality concerns the outer operator applications. Namely, letting `x` denote `dotp bL (Rqv2Rpv s'L) + Rq2Rp h1 + scaleR22Rp (m_decode (if uL then result_R.`2 else result_R.`1))`, the left-hand side becomes `scaleRp2R2t x`, while the right-hand side reduces to `scaleRppq2R2t (scaleRp2Rppq x)`. In Section 4.3.1, we briefly utilized lemma `scaleRp2Rppq2R2t_comp` to demonstrate EasyCrypt's rendering of goals; to reiterate, this lemma proves `scaleRp2R2t x = scaleRppq2R2t (scaleRp2Rppq x)`, for any x of type `Rp`. Certainly, this suggests that applying `rewrite scaleRp2Rppq2R2t_comp` to the goal from Listing 4.65 generates a goal of which the conclusion constitutes an equation with identical expressions on both sides; the imminent listing shows this is indeed the case.

```
Current goal (remaining: 3)

&m: memory
A : Adversary
&1: memory <Game1(A).main>
&2: memory <Game2(A2(A)).main>
uL: bool
H : uL \in {0,1}
sdL: seed
H0: sdL \in dseed
bL: vector
H1: bL \in dRp_vec
H2: bL \in dRp_vec
result_R: plaintext * plaintext
A_R: (glob A)
s'L: Mat_Rq.vector
H3: s'L \in dsmallRq_vec
```

```
H4: s'L \in dsmallRq_vec
------------------------------------------------------------------------
scaleRppq2R2t (scaleRp2Rppq (dotp bL (Rqv2Rpv s'L) + Rq2Rp h1
  + scaleR22Rp (m_decode (if uL then result_R.`2 else result_R.`1))))
=
scaleRppq2R2t (scaleRp2Rppq (dotp bL (Rqv2Rpv s'L) + Rq2Rp h1
  + scaleR22Rp (m_decode (if uL then result_R.`2 else result_R.`1))))
```

Listing 4.66: Proof Demonstration 12 – Goals After `rewrite` Tactic

Naturally, a goal with a conclusion that equates two identical expressions is solvable with `trivial`; as such, applying `trivial` solves the above goal.

After solving the goal in Listing 4.66, the second goal from Listing 4.64 becomes the primary goal[18]. In this goal's conclusion, the right-hand side of the equality successively applies the `c_decode_g` and `c_encode_g` operators; hence, as before, we progress by applying `rewrite cg_enc_dec_inv`, generating the goal in Listing 4.67.

```
Current goal (remaining: 2)

&m: memory
A : Adversary
&1: memory <Game1(A).main>
&2: memory <Game2(A2(A)).main>
uL: bool
H : uL \in {0,1}
sdL: seed
H0: sdL \in dseed
bL: vector
H1: bL \in dRp_vec
H2: bL \in dRp_vec
result_R: plaintext * plaintext
A_R: (glob A)
s'L: Mat_Rq.vector
H3: s'L \in dsmallRq_vec
H4: s'L \in dsmallRq_vec
------------------------------------------------------------------------
scaleRqv2Rpv (trmx (gen sdL) *^ s'L + h)
=
(scaleRp2Rppq (dotp bL (Rqv2Rpv s'L) + Rq2Rp h1
  + scaleR22Rp (m_decode (if uL then result_R.`2 else result_R.`1))),
  scaleRqv2Rpv (trmx (gen sdL) *^ s'L + h)).`2
```

Listing 4.67: Proof Demonstration 13 – Goals After `rewrite` Tactic

As expected, the application of `rewrite cg_enc_dec_inv` effectively removed the sequential applications of `c_decode_g` and `c_encode_g` from the goal's conclusion. Scrutinizing the resulting goal, we observe that the right-hand side of the conclusion's equality contains a 2-tuple: the first element constitutes `scaleRp2Rppq (dotp bL (Rqv2Rpv s'L) + Rq2Rp h1 + scaleR22Rp (m_decode (if uL then result_R.`2 else result_R.`1)))`, and the second element is `scaleRqv2Rpv (trmx (gen sdL) *^ s'L + h)`. Moreover, denoting this tuple by `t`, we see that the right-hand side of the conclusion's equality reduces to `t.`2`; that is, in actuality, this side of the equality is equal to the second element of `t`, i.e., `scaleRqv2Rpv (trmx (gen sdL) *^ s'L + h)`. Indeed, because this expression is identical to the expression on the left-hand side of the equality, the conclusion of the goal trivially holds. Lastly, since the `trivial` tactic is capable of evaluating the `` `.2 `` operator, the

[18]This goal has an identical context to the primary goal in Listing 4.64.

application of this tactic solves the above goal.

Finally, by solving the goal in Listing 4.67, we have completed the formal verification of the assumption that we aspired to introduce in the beginning of the proof: `eq_pr: Pr[Game1(A).main() @ &m : res] = Pr[Game2(A2(A)).main() @ &m : res]`. As such, the goal presented in Listing 4.68 remains the only goal to solve.

```
Current goal

&m: memory
A : Adversary
eq_pr: Pr[Game1(A).main() @ &m : res] = Pr[Game2(A2(A)).main() @ &m : res]
------------------------------------------------------------------------
`|Pr[Game1(A).main() @ &m : res] - 1%r / 2%r| =
`|Pr[Game2(A2(A)).main() @ &m : res] - 1%r / 2%r|
```

Listing 4.68: Proof Demonstration 14 – Final Goal

Utilizing the introduced assumption, the formal verification of this last goal is relatively straightforward. Specifically, applying `rewrite eq_pr` replaces `Pr[Game1(A).main() @ &m : res]` in the left-hand side of the conclusion's equality by `Pr[Game2(A2(A)).main() @ &m : res]`. After this replacement, both sides of the equality are identical; therefore, applying `trivial` solves the goal, completing the formal verification of the `Step_Game1_Game2` lemma. This concludes the demonstration of the (lemma-)proving process in EasyCrypt.

Chapter 5

Conclusions

In this thesis, we considered the formal verification of (the specification of) the PKE scheme provided in Saber, one of the selected few post-quantum cipher suites currently eligible for potential standardization. More precisely, we endeavored to formally verify the desired security and correctness properties of Saber's PKE scheme in the EasyCrypt tool. For both of these properties, the results of this endeavor are affirmative; that is, these results indicate that Saber's PKE scheme indeed satisfies the desired security and correctness properties.

In the process leading up to the obtained results, we initially analyzed Saber's PKE scheme manually, devising hand-written proofs showing the scheme's possession of the desired security and correctness properties. Concerning the hand-written security proof, we purposely adopted the code-based, game-playing approach to the provable security paradigm since this is the primary proof method supported by EasyCrypt. Furthermore, for utilization in this security proof, we constructed two custom variants of the MLWR hardness assumption, i.e., GMLWR and XMLWR. Justifying the use of these custom hardness assumptions, we showed that, in the ROM, the GMLWR and XMLWR problems are as hard as the MLWR problem. Leveraging these concepts, the resulting security proof relates the IND-CPA security of Saber's PKE scheme to the hardness of the GMLWR and XMLWR problems, claiming that compromising the scheme's IND-CPA security is as hard as solving any of the two problems. In essence, this insinuates that, assuming the MLWR problem is amply hard, Saber's PKE scheme is IND-CPA secure *if* it employs an adequate method of generating pseudorandomness (i.e., *if* it utilizes an adequate instantiation of `gen`).

Regarding the hand-written correctness proof, we first composed an alternative specification for Saber's PKE scheme and proved it equivalent to its original counterpart. The principal reason for establishing this alternative specification was the simplification of both the manual analysis and the corresponding formal verification effort. Specifically, we accomplished this by minimizing the algebraic structures utilized in the scheme's computations. This change simplified the manual analysis by diminishing the need to be aware of the interpretation of the considered elements; moreover, this alteration reduced the complexity of the corresponding formal verification effort by decreasing the number of necessary types and type conversions in the performed operations. Employing this alternative specification, we considered the correctness of Saber's PKE scheme with respect to two slightly different definitions: standard correctness, the definition utilized in Saber's original analysis, and FO-correctness, the definition used in (the variant of) the FO transformation applied to Saber's PKE scheme to produce Saber's KEM. Modeling these correctness definitions as probabilistic programs, we attested that they are equivalent in the context of Saber's PKE scheme. Utilizing the above artifacts and results, the constructed correctness proof demonstrates that if the output distribution of the utilized instantiation of gen (practically) equals the appropriate uniform distribution, the standard correctness of Saber's PKE scheme is (almost) accurately computed by Saber's script. Assuming a proper instantiation of gen, this suggests that for any parameter set, the concrete probability produced by Saber's script (nearly) matches the actual standard correctness of Saber's PKE scheme; indeed, this includes the concrete probabilities for Saber's customary parameter sets, publicized in the original paper and official specification. Additionally, the correctness proof shows that the scheme's FO-correctness is invariably equal to its standard correctness and, hence, can (nearly) be computed through Saber's script as well. Consequently, the properties of Saber's KEM, which are directly dependent on the FO-correctness of Saber's PKE scheme, can straightforwardly be evaluated by utilizing the probabilities produced per Saber's script.

The actual formal verification effort regarding the desired security and correctness properties of Saber's PKE scheme closely resembled the scheme's manual analysis. Certainly, as alluded to above, the devised hand-written proofs are structured with the purpose of facilitating the formal verification process in EasyCrypt; as such, accurately leveraging the appropriate features, we were able to formalize the hand-written proofs in EasyCrypt through a virtually literatim translation. As aforementioned, the results of this effort are assenting, i.e., they imply that Saber's PKE scheme indeed possesses the desired security and correctness properties.

Finally, due to the affirmative results of the formal verification effort carried out in this thesis, we have established a higher level of confidence in the security and correctness of Saber's PKE scheme; in turn, assuming the validity of the relevant FO transformation, these results have additionally increased the confidence in the security and correctness of Saber's KEM. This latter increase is a consequence of the direct relation between the properties of Saber's KEM and Saber's PKE scheme, induced by the relevant FO transformation. As a result of the formal verification endeavor performed in independent previous work, the validity assumption concerning this transformation seems quite reasonable [26]. Altogether, this thesis should provide the cryptographic community with more certainty regarding Saber's suitability for standardization as a post-quantum cipher suite.

5.1 Future Work

Concerning potential future work, we recognize several ways in which this thesis's work can be utilized or extended to garner further results contributory to the process of appropriately standardizing post-quantum cryptography, particularly regarding Saber.

First, the results of other relevant formal verification research can be integrated with the results of this thesis. However, if this other research utilizes tools different from EasyCrypt, a sound justification that substantiates the compatibility of the results may be desirable to reduce the likelihood of an unwarranted integration. Considering the presumable complexity of the syntactical and semantical disparities between tools, this justification might be rather intricate and difficult to (manually) verify. Evidently, this intricacy increases the possibility of creating an erroneous justification; if feasible, formally verifying the justification might partially attenuate this problem. As explicated in Section 1.4, an example of an integration that would benefit from such a justification involves the results of this thesis and the result of the above-mentioned formal verification effort regarding the relevant FO transformation.

Second, the relevant FO transformation can be formally verified generically in EasyCrypt; that is, the application of this transformation on a generic PKE scheme conforming to the minimal requirements can be formally verified to produce a (generic) KEM satisfying the properties that the transformation attempts to guarantee. Subsequently, Saber's KEM can be formally verified by instantiating the generic PKE scheme with Saber's PKE scheme; certainly, this is essentially equivalent to formally verifying whether Saber's PKE scheme satisfies the minimal requirements defined for the generic PKE scheme. Naturally, rather than via the instantiation of the generic formal verification, Saber's KEM can also be formally verified directly; indeed, this effectively comes down to formally verifying the relevant FO transformation in the concrete case of its application to Saber's PKE scheme. Patently, the disadvantage of this concrete approach relative to the generic approach is the incapability of reusing the constructed formal verification proof in the context of other KEMs produced per this transformation. Since there exist a multitude of relatively prevalent KEMs generated in this manner, this disadvantage is quite significant.

Lastly, leveraging the results of this thesis, the Jasmin framework can be employed in conjunction with EasyCrypt to construct efficient and formally verified implementations of Saber's schemes. This contributes to the process of standardizing post-quantum cryptography because such high-speed and high-assurance implementations are coveted for deployment in the real world. Considering an implementation of Saber's PKE scheme in Jasmin's programming language, e.g., one of the implementations I created in my research internship [20], the results of this thesis can be utilized in the formal verification of this implementation's functional correctness. Namely, the functional correctness tool from the Jasmin framework generates EasyCrypt code that facilitates the formal verification of an implementation's functional correctness. However, this requires a formalization of the considered construction's specification in EasyCrypt; indeed, for Saber's PKE scheme, this formalization has been established in this thesis. Furthermore, since, by virtue of this thesis, the specification of Saber's PKE scheme is formally verified to satisfy the desired security and correctness properties, formally verifying the functional correctness of an implementation with respect to this specification directly implies that the implementation also possesses these properties.

As mentioned in Section 1.3, this thesis is part of an encompassing project that, following a general formal verification process based on EasyCrypt and Jasmin, seeks to formally verify the specifications of Saber's PKE scheme and KEM, as well as construct formally verified, optimized implementations of these schemes. As such, in addition to the work performed in this thesis, this encompassing project essentially intends to carry out the latter two suggestions for future work presented above.

Bibliography

[1] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography.* Chapman & Hall/CRC, 3rd edition, December 2020.

[2] Lynn Batten. *Public Key Cryptography: Applications and Attacks.* IEEE Press Series on Information and Communication Networks Security. John Wiley & Sons, 1st edition, January 2013.

[3] Mika Hirvensalo. *Quantum Computing.* Natural Computing Series. Springer-Verlag Berlin Heidelberg, 2nd edition, December 2003.

[4] Song Y. Yan. *Quantum Attacks on Public-Key Cryptosystems.* Springer, Boston, MA, 1st edition, April 2013.

[5] Emily Grumbling and Mark Horowitz. *Quantum Computing: Progress and Prospects.* National Academies of Sciences, Engineering, and Medicine. The National Academies Press, 1st edition, April 2019.

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

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

[8] Fabio Cavaliere, John Mattsson, and Ben Smeets. The security implications of quantum cryptography and quantum computing. *Network Security*, 2020(9):9–15, September 2020.

[9] Andy Majot and Roman Yampolskiy. Global catastrophic risk and security implications of quantum computers. *Futures*, 72:17–26, September 2015. Confronting Future Catastrophic Threats To Humanity.

[10] Michele Mosca. Cybersecurity in an era with quantum computers: Will we be ready? *IEEE Security & Privacy*, 16:38–41, September 2018.

[11] Vikas Hassija, Vinay Chamola, Vikas Saxena, Vaibhav Chanana, Prakhar Parashari, Shahid Mumtaz, and Mohsen Guizani. Present landscape of quantum computing. *IET Quantum Communication*, 1(2):42–48, December 2020.

[12] Lidong Chen. Cryptography standards in quantum time: New wine in an old wineskin? *IEEE Security & Privacy*, 15(4):51–57, July 2017.

[13] National Institute of Standards and Technology. Post-Quantum Cryptography | CSRC. `https://csrc.nist.gov/projects/post-quantum-cryptography`, 2016. Accessed: July 28, 2021.

[14] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. Saber: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM. In Antoine Joux, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *AFRICACRYPT 18: 10th International Conference on Cryptology in Africa*, volume 10831 of *Lecture Notes in Computer Science*, pages 282–305. Springer, Heidelberg, May 2018.

[15] Abhishek Banerjee, Chris Peikert, and Alon Rosen. Pseudorandom functions and lattices. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, volume 7237 of *Lecture Notes in Computer Science*, pages 719–737. Springer, Heidelberg, April 2012.

[16] Joël Alwen, Stephan Krenn, Krzysztof Pietrzak, and Daniel Wichs. Learning with rounding, revisited - new reduction, properties and applications. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 57–74. Springer, Heidelberg, August 2013.

[17] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. Why does cryptographic software fail? a case study and open problems. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, APSys '14, pages 1–7. Association for Computing Machinery, June 2014.

[18] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. Easycrypt: A tutorial. In Alessandro Aldini, Javier Lopez, and Fabio Martinelli, editors, *Foundations of Security Analysis and Design VII: FOSAD 2012/2013 Tutorial Lectures*, pages 146–166, Cham, 2014. Springer International Publishing.

[19] 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. Springer, Heidelberg, May / June 2006.

[20] Matthias Meijers. A high-assurance and high-speed implementation of Saber in Jasmin. Internship Report, Eindhoven University of Technology, December 2020.

[21] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 1807–1823. ACM Press, October / November 2017.

[22] Gilles Barthe, Sunjay Cauligi, Benjamin Gregoire, Adrien Koutsos, Kevin Liao, Tiago Oliveira, Swarn Priya, Tamara Rezk, and Peter Schwabe. High-assurance cryptography software in the spectre era. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1884–1901. IEEE Computer Society, May 2021.

[23] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, and Damien Stehlé. CRYSTALS – Kyber: a CCA-secure module-lattice-based KEM. In *2018 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 353–367. IEEE Computer Society, April 2018.

[24] National Institute of Standards and Technology. Third PQC Standardization Conference. `https://csrc.nist.gov/Events/2021/third-pqc-standardization-conference`, 2021. Accessed: July 31, 2021.

[25] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In Yael Kalai and Leonid Reyzin, editors, *TCC 2017: 15th Theory of Cryptography Conference, Part I*, volume 10677 of *Lecture Notes in Computer Science*, pages 341–371. Springer, Heidelberg, November 2017.

[26] Dominique Unruh. Post-quantum verification of fujisaki-okamoto. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2020*, pages 321–352. Springer International Publishing, December 2020.

[27] David S. Dummit and Richard M. Foote. *Abstract Algebra.* Wiley, 3rd edition, June 2003.

[28] Joseph Gallian. *Contemporary Abstract Algebra.* Cengage, 9th edition, January 2017.

[29] Kenneth Ireland and Michael Rosen. *A Classical Introduction to Modern Number Theory.* Springer-Verlag New York, 2nd edition, September 1990.

[30] Harold Davenport. *The Higher Arithmetic: An Introduction to the Theory of Numbers.* Cambridge University Press, 8th edition, October 2008.

[31] Robert B. Ash. *Basic Probability Theory.* Dover Publications, 1st edition, June 2008.

[32] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange - A new hope. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 2016: 25th USENIX Security Symposium*, pages 327–343. USENIX Association, August 2016.

[33] Jan-Pieter D'Anvers. *Design and Security Analysis of Lattice-Based Post-Quantum Encryption.* Ph.D. Dissertation, KU Leuven Arenberg Doctoral School, May 2021.

[34] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography.* Discrete Mathematics and Its Applications. CRC Press, 1st edition, October 1996.

[35] Daniele Micciancio and S. Goldwasser. *Complexity of Lattice Problems: A Cryptographic Perspective.* The Springer International Series in Engineering and Computer Science. Springer US, 1st edition, March 2002.

[36] 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`.

[37] Yehuda Lindell. How to simulate it – a tutorial on the simulation proof technique. In Yehuda Lindell, editor, *Tutorials on the Foundations of Cryptography: Dedicated to Oded Goldreich*, pages 277–346. Springer International Publishing, April 2017.

[38] Oded Regev. The learning with errors problem (invited survey). In *2010 IEEE 25th Annual Conference on Computational Complexity*, pages 191–204. IEEE Computer Society, June 2010.

[39] Chris Peikert. A decade of lattice cryptography. *Foundations and Trends in Theoretical Computer Science*, 10(4):283–424, March 2016.

[40] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *37th Annual ACM Symposium on Theory of Computing*, pages 84–93. ACM Press, May 2005.

[41] Chris Peikert. Public-key cryptosystems from the worst-case shortest vector problem: extended abstract. In Michael Mitzenmacher, editor, *41st Annual ACM Symposium on Theory of Computing*, pages 333–342. ACM Press, May / June 2009.

[42] Zvika Brakerski, Adeline Langlois, Chris Peikert, Oded Regev, and Damien Stehlé. Classical hardness of learning with errors. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *45th Annual ACM Symposium on Theory of Computing*, pages 575–584. ACM Press, June 2013.

[43] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23. Springer, Heidelberg, May / June 2010.

[44] Chris Peikert, Oded Regev, and Noah Stephens-Davidowitz. Pseudorandomness of ring-LWE for any ring and modulus. In Hamed Hatami, Pierre McKenzie, and Valerie King, editors, *49th Annual ACM Symposium on Theory of Computing*, pages 461–473. ACM Press, June 2017.

[45] Martin R. Albrecht and Amit Deo. Large modulus ring-LWE $\geq$ module-LWE. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 267–296. Springer, Heidelberg, December 2017.

[46] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography*, 75(3):565–599, June 2015.

[47] Katharina Boudgoust, Corentin Jeudy, Adeline Roux-Langlois, and Weiqiang Wen. Towards classical hardness of module-LWE: The linear rank case. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2020, Part II*, volume 12492 of *Lecture Notes in Computer Science*, pages 289–317. Springer, Heidelberg, December 2020.

[48] Jacob Alperin-Sheriff and Daniel Apon. Dimension-preserving reductions from LWE to LWR. Cryptology ePrint Archive, Report 2016/589, 2016. `https://eprint.iacr.org/2016/589`.

[49] Gilles Barthe, Juan Manuel Crespo, Benjamin Grégoire, César Kunz, and Santiago Zanella Béguelin. Computer-aided cryptographic proofs. In Lennart Beringer and Amy Felty, editors, *Interactive Theorem Proving*, pages 11–27. Springer Berlin Heidelberg, August 2012.

[50] 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. Springer, Heidelberg, August 2011.

[51] Ran Canetti, Alley Stoughton, and Mayank Varia. EasyUC: Using EasyCrypt to mechanize proofs of universally composable security. In Stephanie Delaune and Limin Jia, editors, *CSF 2019: IEEE 32st Computer Security Foundations Symposium*, pages 167–183. IEEE Computer Society Press, 2019.

[52] Dominique Unruh. Quantum relational hoare logic. *Proceedings of the ACM on Programming*

Languages, 3(POPL):1–31, January 2019.

[53] Dan Boneh, Özgür Dagdelen, Marc Fischlin, Anja Lehmann, Christian Schaffner, and Mark Zhandry. Random oracles in a quantum world. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology – ASIACRYPT 2011*, volume 7073 of *Lecture Notes in Computer Science*, pages 41–69. Springer, Heidelberg, December 2011.

[54] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella-Béguelin. Formal certification of code-based cryptographic proofs. In *36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009*, pages 90–101. Association for Computing Machinery, January 2009.