Formal Verification of Saber's Public-Key Encryption Scheme in EasyCrypt

Andreas Hülsing[1], Matthias Meijers[1], and Pierre-Yves Strub[2]

[1] Eindhoven University of Technology, The Netherlands
[2] Meta, France
fv-saber-pke@mmeijers.com

Abstract In this work, we consider the formal verification of the public-key encryption scheme of Saber, one of the selected few post-quantum cipher suites currently considered for potential standardization. We formally verify this public-key encryption scheme's IND-CPA security and δ-correctness properties, i.e., the properties required to transform the public-key encryption scheme into an IND-CCA2 secure and δ-correct key encapsulation mechanism, in EasyCrypt. To this end, we initially devise hand-written proofs for these properties that are significantly more detailed and meticulous than the presently existing proofs. Subsequently, these hand-written proofs serve as a guideline for the formal verification. The results of this endeavor comprise hand-written and computer-verified proofs which demonstrate that Saber's public-key encryption scheme indeed satisfies the desired security and correctness properties.

Keywords: Formal Verification · Saber · EasyCrypt

1 Introduction

In 1994, Shor showed how to efficiently solve the integer factorization and discrete logarithm problems on a sufficiently powerful quantum computer [Sho94]. Consequently, since contemporary public-key cryptography is predominantly based on the hardness of these problems, the advent of such quantum computers would enable adversaries to compromise the security provided by this type of cryptography [Yan13]. Although it is not entirely clear when the first sufficiently powerful quantum computer will be operational, the progress made hitherto, the currently remaining challenges, and the amount of interest in this topic suggest that this

Andreas Hülsing and Matthias Meijers are funded by an NWO VIDI grant (Project No. VI.Vidi.193.066). At the time of writing, Pierre-Yves Strub was at Institut Polytechnique de Paris, France, and was partially supported by the ERC Advanced Grant *Procontra* (ID: 885666). Date: 2022-06-23.

might well transpire in the near future [Mos18]. As such, a timely replacement of the contemporary public-key cryptography by quantum-resistant alternatives is imperative.

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; this competition is in its final stage, leaving only a selected few of the best candidates. One of these candidates is Saber, a suite of post-quantum cryptographic constructions for public-key encryption and key establishment [DKRV18]. In particular, Saber comprises an IND-CCA2 secure key encapsulation mechanism (KEM) which is the suite's principal scheme of interest due to NIST announcing that they will exclusively standardize KEMs as stand-alone constructions for key encipherment. Saber's KEM is obtained by applying a variant of the Fujisaki-Okamoto (FO) transform on an IND-CPA secure public-key encryption (PKE) scheme. Given that the FO transform has already been analyzed previously [Unr20], this work directs its attention to Saber's PKE scheme, analyzing the claimed security and correctness properties of this PKE scheme necessary to transform it into a secure and (sufficiently) correct KEM.

Historically, (the specifications of) cryptographic constructions have been demonstrated to possess their desired properties through 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 a result, devising these constructions and carrying out the corresponding hand-written proofs has become substantially more challenging. Numerous examples 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 [KM19]. These instances exemplify the intricacy of contriving and verifying cryptographic constructions and their proofs. In addition, even if (the specification of) a cryptographic construction and its proof are entirely correct, implementation flaws may invalidate any of the construction's properties and guarantees. Once again, ample examples exist of this phenomenon, signifying the complexity of constructing and verifying cryptographic implementations [LCWZ14].

The complexity issues associated with devising cryptographic constructions, proofs, and implementations partially induced the inception of the scientific field of computer-aided cryptography. As its name suggests, this field endeavors to devise computer-assisted methods for constructing and verifying cryptography [BBB+21]. The purpose of these computer-assisted methods is to reduce the complexity of the manual labor required in the construction and verification process while consistently enforcing a high level of rigor. This increases the confidence in the cryptographic specifications and implementations that are devised and analyzed in this manner.

Over the years, the research conducted in the field of computer-aided cryptography has produced copious tools and frameworks aimed at the construction and verification of cryptography in a multitude of different ways and con-

texts [BBB+21]. Furthermore, these tools and frameworks have been successfully employed in the construction and verification of increasingly prominent targets; for example, EasyCrypt and Jasmin have been used to construct and verify a functionally correct, constant-time, and efficient implementation of ChaCha20-Poly1305 [ABB+20], CryptoVerif has been used to verify (the specification of) the Hybrid Public-Key Encryption (HPKE) standard [ABH+21], and Tamarin has been used to verify (the specification of) TLS 1.3 [CHH+17]. For a more comprehensive overview and discussion, see [BBB+21]. Based on the context in which Saber's PKE scheme and the considered proofs manifest themselves, the tool of choice for this work is EasyCrypt. Namely, EasyCrypt adopts the code-based approach to provable security, modeling common security-related concepts, such as security notions and hardness assumptions, as well-defined probabilistic programs [BCG+12,BR06]. Additionally, the tool's higher-order ambient logic, standard library, and other built-in mechanisms allow for extensive and (partially) automated mathematical reasoning. Altogether, these features facilitate a natural formalization and manageable verification of the considered proofs for the desired security and correctness properties of Saber's PKE scheme.

Our Contribution. This work considers the formal verification of the desired security and correctness properties of Saber's PKE scheme in EasyCrypt. More precisely, for security, this concerns the IND-CPA property based on the assumed computational hardness of the Module Learning With Rounding (MLWR) problem; for correctness, this concerns the δ-correctness property defined in [HHK17]. In addition to discussing the EasyCrypt-related material, we also present the hand-written proofs devised specifically for this formal verification endeavor; indeed, compared to the currently existing hand-written proofs, these are significantly more detailed and, hence, less ambiguous.

The purpose of this work is to establish a higher level of confidence in the security and correctness of Saber's PKE scheme and, by extension, the KEM obtained from applying the relevant variant of the FO transform discussed in [HHK17]. In accordance with the above-mentioned properties that we consider for Saber's PKE scheme, this relevant variant of the FO transform is $\mathrm{FO}^{\not\perp}$, i.e., the variant that transforms an IND-CPA secure and δ-correct PKE scheme into an IND-CCA2 secure and δ-correct KEM[3]. Naturally, this ultimately serves the purpose of assisting the cryptographic community in making a well-informed decision regarding the standardization of post-quantum cryptography. Thus far, no other formal verification efforts regarding Saber's schemes have been carried out (or, at least, no such endeavors are publicly known).

As an additional contribution, we construct and extend several EasyCrypt libraries with the generic definitions and properties of multiple concepts that are used in the formal verification of Saber's PKE scheme. In particular, we create a library that generically defines the structure and behavior of polynomial quotient rings. Moreover, we extend several existing libraries with results regarding

[3] Although the original paper of Saber states that Saber's KEM is constructed through this variant, the specification of Saber's KEM shows that, technically, a subtly different transform has been used [DKRV18,DHK+21].

distributions over integer rings and polynomials. Due to their abstractness, these libraries are reusable and can be employed in the analysis of other cryptographic systems that use the same or similar mathematical structures, e.g., other lattice-based cryptographic systems. Although we do not explicitly present them here, these libraries can be found in the code associated with this work or in the standard library of EasyCrypt.

Full-Fledged Formal Verification of Saber-Based KEM. Albeit the work presented in this paper has merit on its own, a natural addition would be to formally verify the relevant variant of the FO transform and, thus, obtain a complete formal verification of (the specification of) a secure and (sufficiently) correct KEM based on Saber's PKE. Actually, the formal verification of this transformation has already been performed in previous independent work by Unruh [Unr20]. While certainly contributory and valuable, Unruh's work employs the qRHL tool; indeed, this unfortunately implies that combining his work with this work requires manual reasoning about the compatibility of results obtained through different tools with vastly different syntaxes and semantics. Naturally, such reasoning might be error-prone and, therefore, not ideal. As such, a more robust alternative for extending this work would be to perform the formal verification of the FO transform in EasyCrypt, enabling the direct verification of the compatibility of the results within EasyCrypt. However, this goes beyond the scope of this work.

Overview. The remainder of this paper is structured as follows. First, Section 2 introduces the notation utilized throughout this paper and restates the specification of Saber's PKE scheme. Second, Section 3 discusses the hand-written proof and formal verification of the security property of Saber's PKE scheme. Finally, Section 4 is analogous to Section 3 but, instead of the security property, considers the correctness property of Saber's PKE scheme.

2 Preliminaries

Notation. 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 $\mathbb{Z}_q$. Additionally, we define R_q to be $\mathbb{Z}_q[X]/(X^n+1)$, where $n = 2^{\epsilon_n}$ for some $\epsilon_n \in \mathbb{N}$. As a final extension, we let $R_q^{m \times n}$ stand for the R_q-module of $m \times n$-dimensional matrices over R_q.

Second, for any natural numbers $0 < p$ and $0 < q$, we define a "modular scaling and flooring" function $\lfloor \cdot \rceil_{q \to p} : \mathbb{Z}_q \to \mathbb{Z}_p$, based on the closely related function defined in [BPR12]. Furthermore, we straightforwardly define coefficient-wise and entry-wise extensions of this function to polynomials and vectors/matrices, respectively. Given any $x \in \mathbb{Z}_q$, the modular scaling and flooring function (for some valid p and q) computes the image corresponding to x as follows.

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

Although not explicitly stated, $\frac{p}{q} \cdot x$ is computed over the field of real numbers; to this end, the function uses the obvious interpretations of p, q, and x as real numbers. Notice that when p and q are both powers of two, the modular scaling and flooring operator is equivalent to a regular bit-shift (to the left if $p > q$ and to the right if $p < q$).

Third, for any natural numbers $0 < p$ and $0 < q$, we define a "modular scaling and rounding" function $\lfloor \cdot \rceil_{q \to p} : \mathbb{Z}_q \to \mathbb{Z}_p$ that is identical to the modular scaling and flooring function (for the same p and q), except that it uses rounding instead of flooring. Moreover, this operator is extended similarly to the modular scaling and flooring operator.

Fourth, analogously to the above modular scaling functions, we extend the modular reduction function coefficient-wise and entry-wise to polynomials and vectors/matrices, respectively.

Fifth, we denote the uniform distribution by $\mathcal{U}$ and the centered binomial distribution by β_μ. Furthermore, for distribution $\chi \in \{\mathcal{U}, \beta_\mu\}$ and R_q-module $R_q^{m \times n}$ (as defined above), we use $\chi(R_q^{m \times n})$ to signify the distribution over matrices from $R_q^{m \times n}$ that arises when (each coefficient of) every entry is distributed according to χ.

Finally, we typeset regular (i.e., non-vector and non-matrix) elements with lowercase, italic letters; vectors with lowercase, boldface letters; and matrices with uppercase, boldface letters. Additionally, sampling from a distribution χ and storing the result in x is written as $x \leftarrow\!\$\ \chi$. Lastly, in bit strings, we denote (sub)strings of n consecutive 0 or (n consecutive) 1 bits by 0^n or 1^n, respectively.

Saber's PKE. In this work, we adopt the specification of Saber's PKE scheme that is provided in the original paper [DKRV18]. For intelligibility purposes, Algorithm 1, Algorithm 2, and Algorithm 3 respectively restate the specifications of the scheme's key generation, encryption[4], and decryption algorithms utilizing the above-introduced notation. In these specifications, identifiers gen, l, t, p, q, h_1, h_2, and $\mathbf{h}$ refer to the same function, parameters, and constants as in the original specifications. Particularly, recall that $t = 2^{\epsilon_t}$, $p = 2^{\epsilon_p}$, and $q = 2^{\epsilon_q}$ for some $\epsilon_t, \epsilon_p, \epsilon_q \in \mathbb{N}$ such that $0 < \epsilon_t + 1 < \epsilon_p < \epsilon_q$; furthermore, $h_1 = \sum_{i=0}^{n-1} \frac{q}{2 \cdot p} \cdot X^i$ and $h_2 = \sum_{i=0}^{n-1} (\frac{p}{4} - \frac{p}{4 \cdot t}) \cdot X^i$, both elements of R_q, while $\mathbf{h}$ is defined as the vector with all entries equal to h_1. Crucially, as a consequence of these definitions, we have that for all $x \in \mathbb{Z}_q$, $\lfloor x \rceil_{q \to p} = \lfloor x + h_1 \rfloor_{q \to p}$; by extension, $\lfloor \mathbf{v} \rceil_{q \to p} = \lfloor \mathbf{v} + \mathbf{h} \rfloor_{q \to p}$ for all $\mathbf{v} \in R_q^{l \times 1}$.

Throughout the remainder, Saber's PKE scheme will be referred to by the identifier Saber.PKE; the scheme's algorithms will be denoted by their respective procedure identifiers provided in the specifications.

3 Security

In this section, we cover Saber.PKE's security property by discussing the essential parts of the hand-written proof and the corresponding formal verification

[4] Note that the message $m \in \{0,1\}^n$ is implicitly encoded as an element of R_2 by dedicating a separate coefficient to each bit.

Algorithm 1 Saber's Key Generation Algorithm

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

Algorithm 2 Saber's Encryption Algorithm

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

Algorithm 3 Saber's Decryption Algorithm

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

in EasyCrypt. Due to space considerations, some less informative material is omitted from this section and provided in the appendix instead. At the relevant places, we explicitly mention which material this concerns.

Notion and Assumptions. Prior to discussing the actual hand-written proof, we introduce the relevant security notion and hardness assumptions. Here, we do not yet present any formalizations in EasyCrypt; instead, we postpone this to the relevant places in the discussion concerning the formal verification.

Foremost, we reiterate that Saber.PKE attempts to achieve the IND-CPA security notion based on the assumed computational hardness of the MLWR problem; the specifications of the corresponding games are respectively provided in Figure 1 and Figure 2. Then, the advantage of an adversary $\mathcal{A} = (\mathsf{P}, \mathsf{D})$ against $\text{Game}^{\text{IND-CPA}}_{\mathcal{A},\text{Saber.PKE}}$ is defined as follows.

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

Moreover, the advantage of an adversary $\mathcal{A}$ against $\text{Game}^{\text{MLWR}}_{\mathcal{A},m,l,\mu,q,p}(u)$ is defined as given below.

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

Rather than directly employing the MLWR game, our proof utilizes two variant games, GMLWR and XMLWR, that (partly) generate the matrix $\mathbf{A}$

$\text{Game}^{\text{IND-CPA}}_{\mathcal{A},\text{Saber.PKE}}$

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 : $u' \leftarrow \mathcal{A}.\mathsf{D}(c)$
6 : **return** $(u' = u)$

Figure 1. The IND-CPA Game for Saber.PKE

$\text{Game}^{\text{MLWR}}_{\mathcal{A},m,l,\mu,q,p}(u)$

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 2. The MLWR Game

through a function instead of randomly sampling it. In case this function is a random oracle, these variant games are at least as hard as the MLWR game. This approach makes for a more general security theorem that holds for all valid instantiations of the gen function (and the parameters) of Saber; additionally, this enables us to separately analyze the utilized hardness assumptions in the random oracle model while staging the security proof in the standard model. The specifications of the GMLWR and XMLWR games are respectively given in Figure 3 and Figure 4; notice that the gen parameter in these games is expected to be a mathematical function mapping from bit strings of length 256 to matrices from $R_q^{l\times l}$ (where q and l are two of the other parameters of the games), similar to the identically named function used in Saber.PKE. The advantages of adversaries against these games are defined analogously to the advantage of adversaries against the MLWR game.

$\text{Game}^{\text{GMLWR}}_{\mathcal{A},\mathsf{gen},l,\mu,q,p}(u)$

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. The GMLWR Game

$\text{Game}^{\text{XMLWR}}_{\mathcal{A},\mathsf{gen},l,\mu,q,p}(u)$

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 4. The XMLWR Game

Considering $\text{Game}^{\text{GMLWR}}_{\mathcal{A},\mathsf{gen},l,\mu,q,p}(u)$ and $\text{Game}^{\text{XMLWR}}_{\mathcal{A},\mathsf{gen},l,\mu,q,p}(u)$, we can rather easily observe that if gen is a random oracle, these (instances of) games are at least as hard as $\text{Game}^{\text{MLWR}}_{\mathcal{A},l,l,\mu,q,p}(u)$ and $\text{Game}^{\text{MLWR}}_{\mathcal{A},l+1,l,\mu,q,p}(u)$, respectively. Naturally, these observations can be formalized in random oracle model proofs. In fact, we constructed these proofs, subsequently carrying out their formal verification in EasyCrypt. Since they are relatively simple, we omit these proofs and their formal verification in this discussion; instead, we provide them in Appendix A. Nevertheless, here we do note that these random oracle model proofs exclusively employ history-free reductions, ensuring the validity of these proofs in the quantum setting [BDF+11]. Unfortunately, at the time of performing the formal verification, EasyCrypt did not provide the features necessary to formally verify the soundness of this argument[5].

Hand-Written Proof. Utilizing the above-introduced security notion and hardness assumptions, we devise a code-based, game-playing proof of Saber.PKE's security in the standard model. The security theorem we aim to prove is the following. Here, gen, q, p, t, l, and μ refer to the previously introduced function and parameters of Saber; as such, these are assumed to satisfy the corresponding requirements imposed by Saber.

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 running time as $\mathcal{A}$, such that*

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

Conceptually, the proof of the above theorem is similar to the security proof concerning Saber's key exchange scheme given in the original paper [DKRV18]; in particular, between the proofs, the structures of the game sequences are quite alike, and the justifications of the steps within the proofs are primarily based on the same reasoning. Nevertheless, as aforementioned, the security proof presented in this work is significantly more detailed and meticulous. While constructing such a proof has merit on its own, the primary rationale for this is that it facilitates the subsequent formal verification in EasyCrypt. Namely, the formal verification enforces a high level of rigorousness and granularity on the proof; already having a detailed hand-written proof as a reference eases this process substantially. Naturally, knowing this hand-written proof similarly helps comprehend the material of the corresponding formal verification. For this reason, we cover the hand-written proof that we devised before advancing to the discussion on the formal verification.

The ensuing proof consists of a sequence of five games, depicted in Figure 5; for each game, the statements that differ from the preceding game are highlighted with a gray background. In this sequence, the first game arises from replacing the procedure identifiers in $\text{Game}^{\text{IND-CPA}}_{\mathcal{A},\text{Saber.PKE}}$ by the corresponding specifications; the remainder of the games are slight variations of this initial

[5] However, since then, these features have been implemented and integrated into EasyCrypt, making the formal verification of the validity of the random oracle model proofs in the quantum setting a potential objective for future work [BBF+21].

game. As such, for $\text{Game}^i_{\mathcal{A}}$ $(0 \leq i \leq 4)$, the advantage of $\mathcal{A}$ is defined analogously to $\mathsf{Adv}^{\text{IND-CPA}}_{\text{Saber.PKE}}(\mathcal{A})$; we denote this advantage by $\mathsf{Adv}^i(\mathcal{A})$. We now bound the difference in advantages between any two consecutive games from the game sequence.

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

Considering the difference between $\text{Game}^0_{\mathcal{A}}$ and $\text{Game}^1_{\mathcal{A}}$, we can see that the pair $(\text{seed}_{\mathbf{A}}, \mathbf{b})$ in $\text{Game}^0_{\mathcal{A}}$ is constructed identically to the pair $(\text{seed}_{\mathbf{A}}, \mathbf{b}_0)$ in $\text{Game}^{\text{GMLWR}}_{\mathcal{A},\mathsf{gen},l,\mu,q,p}(u)$; contrarily, in $\text{Game}^1_{\mathcal{A}}$, the pair $(\text{seed}_{\mathbf{A}}, \mathbf{b})$ is constructed identically to the pair $(\text{seed}_{\mathbf{A}}, \mathbf{b}_1)$ in $\text{Game}^{\text{GMLWR}}_{\mathcal{A},\mathsf{gen},l,\mu,q,p}(u)$. Consequently, an adversary $\mathcal{A}$ that is able to distinguish between these two games can be used to construct an adversary $\mathcal{B}_0^{\mathcal{A}}$ against the corresponding instance of the GMLWR game. Figure 6 provides such a reduction adversary.

Based on the reduction adversary given in Figure 6, we can deduce that for any given adversary $\mathcal{A}$ against $\text{Game}^0_{\mathcal{A}}$ and $\text{Game}^1_{\mathcal{A}}$, there exists an adversary $\mathcal{B}_0^{\mathcal{A}}$ against the corresponding instance of the GMLWR game such that $\left|\Pr\left[\text{Game}^0_{\mathcal{A}} = 1\right] - \Pr\left[\text{Game}^1_{\mathcal{A}} = 1\right]\right| = \mathsf{Adv}^{\text{GMLWR}}_{\mathsf{gen},l,\mu,q,p}(\mathcal{B}_0^{\mathcal{A}})$. Indeed, this is a consequence of the fact that, from the perspective of $\mathcal{A}$, $\mathcal{B}_0^{\mathcal{A}}(\text{seed}_{\mathbf{A}}, \mathbf{b}_u)$ 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, this can be accomplished by straightforwardly letting $\mathcal{R}^{\mathcal{A}}$ disregard any additional information it receives relative to the information provided to an adversary against the first game. Figure 7 presents this reduction adversary.

Naturally, for the desired equality of advantages to hold, the reduction adversary should, from the perspective of $\mathcal{A}$, perfectly simulate a run of $\text{Game}^1_{\mathcal{A}}$. Since the only difference between the considered games concerns the computation of $\hat{c}$, this wholly depends on the indistinguishability of $\hat{c}$ (from $\text{Game}^1_{\mathcal{A}}$) and $\hat{c}'$ (from $\text{Game}^2_{\mathcal{R}^{\mathcal{A}}}$); the remainder is trivially identical. Therefore, consider $x = v' + \lfloor m_u \rfloor_{2 \to p}$, where v' and m_u are as in $\text{Game}^1_{\mathcal{A}}$ and $\text{Game}^2_{\mathcal{A}}$. Then, because $\epsilon_t + 1 < \epsilon_p$, the modular scaling and flooring operation performed on x in $\text{Game}^1_{\mathcal{A}}$ effectively carries out a right-bit shift of $\epsilon_p - (\epsilon_t + 1)$ bits on each coefficient of x. Consequently, denoting the binary representation of a coefficient of x by $a_{\epsilon_p - 1} \ldots a_0$, the corresponding coefficient of the resulting $\hat{c}$ equals $a_{\epsilon_p - 1} \ldots a_{\epsilon_p - \epsilon_t - 1}$. Similarly, because $\epsilon_p < \epsilon_q$ (which implies $2 \cdot \epsilon_p - \epsilon_q < \epsilon_q$), $\text{Game}^2_{\mathcal{R}^{\mathcal{A}}}$ essentially performs a right-bit shift of $\epsilon_p - (2 \cdot \epsilon_p - \epsilon_q) = \epsilon_q - \epsilon_p$ bits. Certainly, for each coefficient $a_{\epsilon_p - 1} \ldots a_0$ of x, this gives a resulting coefficient $a_{\epsilon_p - 1} \ldots a_{\epsilon_q - \epsilon_p}$ of $\hat{c}$. At this point, employing the assumption stated in the secu-

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

$u \leftarrow\$\ \mathcal{U}(\{0,1\})$
$\mathrm{seed}_{\mathbf{A}} \leftarrow\$\ \mathcal{U}(\{0,1\}^{256})$
$\mathbf{A} \leftarrow \mathsf{gen}(\mathrm{seed}_{\mathbf{A}})$
$\mathbf{s} \leftarrow\$\ \beta_\mu(R_q^{l\times 1})$
$\mathbf{b} \leftarrow \lfloor \mathbf{A}\cdot\mathbf{s}+\mathbf{h}\rfloor_{q\to p}$
$(m_0,m_1) \leftarrow \mathcal{A}.\mathsf{P}((\mathrm{seed}_{\mathbf{A}},\mathbf{b}))$
$\mathbf{s}' \leftarrow\$\ \beta_\mu(R_q^{l\times 1})$
$\mathbf{b}' \leftarrow \lfloor \mathbf{A}^T\cdot\mathbf{s}'+\mathbf{h}\rfloor_{q\to p}$
$v' \leftarrow \mathbf{b}^T\cdot(\mathbf{s}' \bmod p) + (h_1 \bmod p)$
$\hat{c} \leftarrow \lfloor v' + \lfloor m_u \rfloor_{2\to p}\rfloor_{p\to 2\cdot t}$
$u' \leftarrow \mathcal{A}.\mathsf{D}((\hat{c},\mathbf{b}'))$
return $(u'=u)$

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

$u \leftarrow\$\ \mathcal{U}(\{0,1\})$
$\mathrm{seed}_{\mathbf{A}} \leftarrow\$\ \mathcal{U}(\{0,1\}^{256})$
$\mathbf{A} \leftarrow \mathsf{gen}(\mathrm{seed}_{\mathbf{A}})$
Skip
$\mathbf{b} \leftarrow\$\ \mathcal{U}(R_p^{l\times 1})$
$(m_0,m_1) \leftarrow \mathcal{A}.\mathsf{P}((\mathrm{seed}_{\mathbf{A}},\mathbf{b}))$
$\mathbf{s}' \leftarrow\$\ \beta_\mu(R_q^{l\times 1})$
$\mathbf{b}' \leftarrow \lfloor \mathbf{A}^T\cdot\mathbf{s}'+\mathbf{h}\rfloor_{q\to p}$
$v' \leftarrow \mathbf{b}^T\cdot(\mathbf{s}' \bmod p) + (h_1 \bmod p)$
$\hat{c} \leftarrow \lfloor v' + \lfloor m_u \rfloor_{2\to p}\rfloor_{p\to 2\cdot t}$
$u' \leftarrow \mathcal{A}.\mathsf{D}((\hat{c},\mathbf{b}'))$
return $(u'=u)$

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

$u \leftarrow\$\ \mathcal{U}(\{0,1\})$
$\mathrm{seed}_{\mathbf{A}} \leftarrow\$\ \mathcal{U}(\{0,1\}^{256})$
$\mathbf{A} \leftarrow \mathsf{gen}(\mathrm{seed}_{\mathbf{A}})$
Skip
$\mathbf{b} \leftarrow\$\ \mathcal{U}(R_p^{l\times 1})$
$(m_0,m_1) \leftarrow \mathcal{A}.\mathsf{P}((\mathrm{seed}_{\mathbf{A}},\mathbf{b}))$
$\mathbf{s}' \leftarrow\$\ \beta_\mu(R_q^{l\times 1})$
$\mathbf{b}' \leftarrow \lfloor \mathbf{A}^T\cdot\mathbf{s}'+\mathbf{h}\rfloor_{q\to p}$
$v' \leftarrow \mathbf{b}^T\cdot(\mathbf{s}' \bmod p) + (h_1 \bmod p)$
$\hat{c} \leftarrow \lfloor v' + \lfloor m_u \rfloor_{2\to p}\rfloor_{p\to p^2/q}$
$u' \leftarrow \mathcal{A}.\mathsf{D}((\hat{c},\mathbf{b}'))$
return $(u'=u)$

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

$u \leftarrow\$\ \mathcal{U}(\{0,1\})$
$\mathrm{seed}_{\mathbf{A}} \leftarrow\$\ \mathcal{U}(\{0,1\}^{256})$
$\mathbf{A} \leftarrow \mathsf{gen}(\mathrm{seed}_{\mathbf{A}})$
Skip
$\mathbf{b} \leftarrow\$\ \mathcal{U}(R_q^{l\times 1})$
$(m_0,m_1) \leftarrow \mathcal{A}.\mathsf{P}((\mathrm{seed}_{\mathbf{A}},\mathbf{b}))$
$\mathbf{s}' \leftarrow\$\ \beta_\mu(R_q^{l\times 1})$
$\mathbf{b}' \leftarrow \lfloor \mathbf{A}^T\cdot\mathbf{s}'+\mathbf{h}\rfloor_{q\to p}$
$v' \leftarrow \lfloor \mathbf{b}^T\cdot\mathbf{s}' + h_1\rfloor_{q\to p}$
$\hat{c} \leftarrow v' + (\lfloor m_u \rfloor_{2\to p^2/q} \bmod p)$
$u' \leftarrow \mathcal{A}.\mathsf{D}((\hat{c},\mathbf{b}'))$
return $(u'=u)$

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

$u \leftarrow\$\ \mathcal{U}(\{0,1\})$
$\mathrm{seed}_{\mathbf{A}} \leftarrow\$\ \mathcal{U}(\{0,1\}^{256})$
$\mathbf{A} \leftarrow \mathsf{gen}(\mathrm{seed}_{\mathbf{A}})$
Skip
$\mathbf{b} \leftarrow\$\ \mathcal{U}(R_q^{l\times 1})$
$(m_0,m_1) \leftarrow \mathcal{A}.\mathsf{P}((\mathrm{seed}_{\mathbf{A}},\mathbf{b}))$
Skip
$\mathbf{b}' \leftarrow\$\ \mathcal{U}(R_p^{l\times 1})$
$v' \leftarrow\$\ \mathcal{U}(R_p)$
$\hat{c} \leftarrow v' + (\lfloor m_u \rfloor_{2\to p^2/q} \bmod p)$
$u' \leftarrow \mathcal{A}.\mathsf{D}((\hat{c},\mathbf{b}'))$
return $(u'=u)$

Figure 5. Game Sequence of Saber.PKE's Security Proof

$\mathcal{B}_0^{\mathcal{A}}(\text{seed}_{\mathbf{A}}, \mathbf{b}_u)$

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 : $\mathbf{b}' \leftarrow \lfloor \mathbf{A}^T \cdot \mathbf{s}' + \mathbf{h} \rfloor_{q\to p}$
6 : $v' \leftarrow \mathbf{b}_u^T \cdot (\mathbf{s}' \bmod p) + (h_1 \bmod p)$
7 : $\hat{c} \leftarrow \lfloor v' + \lfloor m_w \rfloor_{2\to p} \rfloor_{p\to 2\cdot t}$
8 : $w' \leftarrow \mathcal{A}.\mathsf{D}((\hat{c}, \mathbf{b}'))$;
9 : **return** $(w' = w)$;

Figure 6. Reduction Adversary $\mathcal{B}_0^{\mathcal{A}}$ Against $\text{Game}^{\text{GMLWR}}_{\mathcal{B}_0^{\mathcal{A}},\mathsf{gen},l,\mu,q,p}(u)$

$\mathcal{R}^{\mathcal{A}}.\mathsf{P}((\text{seed}_{\mathbf{A}}, \mathbf{b}))$

1 : **return** $\mathcal{A}.\mathsf{P}((\text{seed}_{\mathbf{A}}, \mathbf{b}))$

$\mathcal{R}^{\mathcal{A}}.\mathsf{D}((\hat{c}, \mathbf{b}'))$

1 : $\hat{c}' \leftarrow \lfloor \hat{c} \rceil_{p^2/q\to 2\cdot t}$
2 : **return** $\mathcal{A}.\mathsf{D}((\hat{c}', \mathbf{b}'))$

Figure 7. Reduction Adversary $\mathcal{R}^{\mathcal{A}}$ Against $\text{Game}^2_{\mathcal{R}^{\mathcal{A}}}$

rity theorem, i.e., $\frac{q}{p} \leq \frac{p}{2\cdot t}$, we can see that the subsequent modular scaling and flooring operation applied by $\mathcal{R}^{\mathcal{A}}$ on $\hat{c}$ carries out an additional right-bit shift of $2 \cdot \epsilon_p - \epsilon_q - \epsilon_t - 1$ bits; in terms of the original x, this operation transforms each coefficient $a_{\epsilon_p - 1} \ldots a_{\epsilon_q - \epsilon_p}$ into $a_{\epsilon_p - 1} \ldots a_{\epsilon_p - \epsilon_t - 1}$, identical to the corresponding coefficient of $\hat{c}$ in $\text{Game}^1_{\mathcal{A}}$. Thus, the $\hat{c}'$ of $\text{Game}^2_{\mathcal{R}^{\mathcal{A}}}$ indeed is indistinguishable from the $\hat{c}$ of $\text{Game}^1_{\mathcal{A}}$; as such, the reduction adversary is correctly constructed and gives the desired equality of advantages.

Step 3: $\text{Game}^2_{\mathcal{A}}$ - $\text{Game}^3_{\mathcal{A}}$. In this step, similarly to the preceding step, we exclusively introduce changes 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 8 provides such a reduction adversary.

$\mathcal{R}^{\mathcal{A}}.\mathsf{P}((\text{seed}_{\mathbf{A}}, \mathbf{b}))$

1 : $\mathbf{b}_p \leftarrow \mathbf{b} \bmod p$
2 : **return** $\mathcal{A}.\mathsf{P}((\text{seed}_{\mathbf{A}}, \mathbf{b}_p))$

$\mathcal{R}^{\mathcal{A}}.\mathsf{D}((\hat{c}, \mathbf{b}'))$

1 : $\hat{c}' \leftarrow \hat{c} \bmod p^2/q$
2 : **return** $\mathcal{A}.\mathsf{D}((\hat{c}', \mathbf{b}'))$;

Figure 8. Reduction Adversary $\mathcal{R}^{\mathcal{A}}$ Against $\text{Game}^3_{\mathcal{R}^{\mathcal{A}}}$

In order to substantiate that the reduction adversary in Figure 8 perfectly simulates a run of $\text{Game}^2_{\mathcal{A}}$ (from the perspective of $\mathcal{A}$), we argue the following points.

- Sampling $\mathbf{b}$ from $\mathcal{U}(R_q^{l\times 1})$ and, subsequently, reducing it modulo p is well-defined and equivalent to sampling $\mathbf{b}$ directly from $\mathcal{U}(R_p^{l\times 1})$. That is, the $\mathbf{b}_p$ that $\mathcal{R}^{\mathcal{A}}$ provides to $\mathcal{A}$ in $\text{Game}^3_{\mathcal{R}^{\mathcal{A}}}$ is indistinguishable from, i.e., identically distributed to, the $\mathbf{b}$ that $\mathcal{A}$ receives in $\text{Game}^2_{\mathcal{A}}$.
- Utilizing the $\hat{c}$ given in $\text{Game}^3_{\mathcal{R}^{\mathcal{A}}}$, the $\hat{c}'$ that $\mathcal{R}^{\mathcal{A}}$ computes (and calls $\mathcal{A}$ with) is indistinguishable from, i.e., identically distributed to, the $\hat{c}$ provided to $\mathcal{A}$ in $\text{Game}^2_{\mathcal{A}}$.

Certainly, since $\mathbf{b}$ and $\hat{c}$ are the only artifacts provided to the adversary that differ between the considered games, the above two points are sufficient to demonstrate that the reduction adversary gives rise to the desired equality of advantages.

Regarding the first point, suppose $\mathbf{b}$ is sampled from $\mathcal{U}(R_q^{l\times 1})$ as in $\text{Game}^3_{\mathcal{R}^{\mathcal{A}}}$. Then, by definition, every coefficient of (each entry of) $\mathbf{b}$ is an element from $\mathbb{Z}_q$. Since $p \mid q$, reduction modulo p is well-defined for each of these coefficients; in turn, the extension of this modular reduction to the complete vector, i.e., $\mathbf{b} \bmod p$, is well-defined as well. Furthermore, since precisely $\frac{q}{p}$ elements from $\mathbb{Z}_q$ map to a specific $x \in \mathbb{Z}_p$ when reduced modulo p, exactly $\frac{q^{n\cdot l}}{p^{n\cdot l}}$ elements from $R_q^{l\times 1}$ map to a specific $\mathbf{v} \in R_p^{l\times 1}$ when reduced modulo p. Therefore, sampling $\mathbf{b}$ from $\mathcal{U}(R_q^{l\times 1})$ and, subsequently, reducing it modulo p is well-defined and results in an element that is uniformly distributed over $R_p^{l\times 1}$.

Concerning the second point, consider the computation of $\mathbf{b}^T \cdot \mathbf{s}' + h_1$ in $\text{Game}^3_{\mathcal{R}^{\mathcal{A}}}$. As a consequence of the previous point, reducing the result of this computation modulo p provides an identical result to the analogous computation in $\text{Game}^2_{\mathcal{A}}$; equivalently, the computation of $\mathbf{b}^T \cdot (\mathbf{s}' \bmod p) + (h_1 \bmod p)$ (from $\text{Game}^2_{\mathcal{A}}$) and $\mathbf{b}^T \cdot \mathbf{s}' + h_1$ (from $\text{Game}^3_{\mathcal{R}^{\mathcal{A}}}$) are equal in the least significant ϵ_p bits. Thus, if $a_{\epsilon_q-1} \dots a_{\epsilon_p-1} \dots a_0$ denotes the binary representation of a coefficient of the computation in $\text{Game}^3_{\mathcal{R}^{\mathcal{A}}}$, then $a_{\epsilon_p-1} \dots a_0$ denotes its counterpart from $\text{Game}^2_{\mathcal{A}}$. Certainly, from this follows that the corresponding coefficient of v' in $\text{Game}^3_{\mathcal{R}^{\mathcal{A}}}$ is computed as $\lfloor a_{\epsilon_q-1} \dots a_{\epsilon_p-1} \dots a_0 \rfloor_{q\to p} = a_{\epsilon_q-1} \dots a_{\epsilon_p-1} \dots a_{\epsilon_q-\epsilon_p}$. Then, denoting a coefficient of m_u by b_0, we can derive that the corresponding coefficients of $\hat{c}$ between the games are identical in their $2\cdot\epsilon_p - \epsilon_q$ least significant bits as follows[6].

$$\begin{aligned}\text{Game}^2_{\mathcal{A}}: \quad & \lfloor a_{\epsilon_p-1} \dots a_0 + \lfloor b_0 \rfloor_{2\to p} \rfloor_{p\to p^2/q} = \\ & \lfloor a_{\epsilon_p-1} \dots a_0 + b_0 0^{\epsilon_p-1} \rfloor_{p\to p^2/q} = \\ & \lfloor (a_{\epsilon_p-1} + b_0) \dots a_0 \rfloor_{p\to p^2/q} =\end{aligned}$$

[6] Notice that in $\text{Game}^3_{\mathcal{R}^{\mathcal{A}}}$, the explicit modular reduction in $v' + (\lfloor m_u \rfloor_{2\to p^2/q} \bmod p)$ is merely used to accentuate the interpretation of $\lfloor m_u \rfloor_{2\to p^2/q}$ as an element of R_p; that is, the modular reduction does not affect the actual value of m_u.

$$\begin{aligned}
&(a_{\epsilon_p-1}+b_0)\ldots a_{\epsilon_q-\epsilon_p}\\
\text{Game}^3_{\mathcal{R}^{\mathcal{A}}}: \; &a_{\epsilon_q-1}\ldots a_{\epsilon_p-1}\ldots a_{\epsilon_q-\epsilon_p} + (\lfloor b_0 \rfloor_{2\to p^2/q} \bmod p) =\\
&a_{\epsilon_q-1}\ldots a_{\epsilon_p-1}\ldots a_{\epsilon_q-\epsilon_p} + 0^{\epsilon_q-\epsilon_p} b_0 0^{2\cdot\epsilon_p-\epsilon_q-1} =\\
&d_{\epsilon_q-1}\ldots d_{\epsilon_p}(a_{\epsilon_p-1}+b_0)a_{\epsilon_p-2}\ldots a_{\epsilon_q-\epsilon_p}
\end{aligned}$$

Here, $(a_{\epsilon_p-1}+b_0)$ represents the (single) bit value resulting from the addition of the a_{ϵ_p-1} and b_0 bits (modulo 2); furthermore, each d_i represents a bit that might be influenced by potential carries. Finally, we see that $\mathcal{R}^{\mathcal{A}}$ indeed correctly computes (and calls $\mathcal{A}$ with) a $\hat{c}'$ that is indistinguishable from the $\hat{c}$ of $\text{Game}^2_{\mathcal{A}}$ by reducing each coefficient of the $\hat{c}$ provided in $\text{Game}^3_{\mathcal{R}^{\mathcal{A}}}$ modulo $\frac{p^2}{q} = 2^{2\cdot\epsilon_p-\epsilon_q}$ and, hence, effectively discarding the $\epsilon_q-\epsilon_p$ most significant bits.

Step 4: $\text{Game}^3_{\mathcal{A}}$ - $\text{Game}^4_{\mathcal{A}}$. For the final step, we change the manner in which $\mathbf{b}'$ and v' are obtained. Namely, instead of computing these values by $\lfloor \mathbf{A}^T\cdot\mathbf{s}'+\mathbf{h} \rfloor_{q\to p}$ and $\lfloor \mathbf{b}^T\cdot\mathbf{s}'+h_1 \rfloor_{q\to p}$, 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, for this reason, is removed from $\text{Game}^4_{\mathcal{A}}$.

In $\text{Game}^3_{\mathcal{A}}$, the tuple $(\text{seed}_{\mathbf{A}}, \mathbf{b}', \mathbf{b}^T, v')$ is constructed identically to the tuple $(\text{seed}_{\mathbf{A}}, \mathbf{b}_0, \mathbf{a}, d_0)$ in $\text{Game}^{\text{XMLWR}}_{\mathcal{A},\mathsf{gen},l,\mu,q,p}(u)$; contrarily, the tuple $(\text{seed}_{\mathbf{A}}, \mathbf{b}', \mathbf{b}^T, v')$ in $\text{Game}^4_{\mathcal{A}}$ is constructed in the same way as the tuple $(\text{seed}_{\mathbf{A}}, \mathbf{b}_1, \mathbf{a}, d_1)$ in $\text{Game}^{\text{XMLWR}}_{\mathcal{A},\mathsf{gen},l,\mu,q,p}(u)$. 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 instance of the XMLWR game. Figure 9 contains such a reduction adversary.

$\mathcal{B}_1^{\mathcal{A}}(\text{seed}_{\mathbf{A}}, \mathbf{b}_u, \mathbf{a}, d_u)$

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 : $\hat{c} \leftarrow d_u + (\lfloor m_w \rfloor_{2\to p^2/q} \bmod p)$
4 : $w' \leftarrow \mathcal{A}.\mathsf{D}((\hat{c}, \mathbf{b}_u))$
5 : **return** $(w = w')$

Figure 9. Reduction Adversary $\mathcal{B}_1^{\mathcal{A}}$ Against $\text{Game}^{\text{XMLWR}}_{\mathcal{B}_1^{\mathcal{A}},\mathsf{gen},l,\mu,q,p}(u)$

Employing the reduction in Figure 9, we can derive a result analogous to the result of the first step. Specifically, for any adversary $\mathcal{A}$ distinguishing between $\text{Game}^3_{\mathcal{A}}$ and $\text{Game}^4_{\mathcal{A}}$, there exists an adversary $\mathcal{B}_1^{\mathcal{A}}$ against $\text{Game}^{\text{XMLWR}}_{\mathcal{B}_1^{\mathcal{A}},\mathsf{gen},l,\mu,q,p}(u)$ such that $\left|\Pr\left[\text{Game}^3_{\mathcal{A}}=1\right] - \Pr\left[\text{Game}^4_{\mathcal{A}}=1\right]\right| = \mathsf{Adv}^{\text{XMLWR}}_{\mathsf{gen},l,\mu,q,p}(\mathcal{B}_1^{\mathcal{A}})$. Certainly, this is due to the fact that, from the perspective of $\mathcal{A}$, $\mathcal{B}_1^{\mathcal{A}}(\text{seed}_{\mathbf{A}}, \mathbf{b}_u, \mathbf{a}, d_u)$ 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 can 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 $\mathrm{Game}^4_{\mathcal{A}}$ must randomly guess the bit u; as a result, for any adversary $\mathcal{A}$, we have $\Pr\left[\mathrm{Game}^4_{\mathcal{A}} = 1\right] = \frac{1}{2}$.

Final Result. Aggregating all results, we can derive the security theorem as follows.

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

In this derivation, the inequality arises from an application of the triangle inequality; the second-to-last equality follows from the result of the first step in the proof; and the last equality holds due to the results of the second, third, and fourth step, as well as the fact that $\Pr\left[\mathrm{Game}^4_{\mathcal{A}'} = 1\right] = \frac{1}{2}$. At last, compressing the above derivation gives the desired result.

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

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.

Formal Verification. Following the hand-written security proof, we discuss several representative parts of the corresponding formal verification in EasyCrypt. Specifically, we examine the formalization of (part of) the fundamental specification; furthermore, we consider the formalization specific to the initial two proof steps and the final result. Notably, we do not cover the concrete proofs of the results in EasyCrypt. This is mainly because the exposition of such technical endeavors would not be meaningful to the current discussion; moreover, this does not take away from the meaningfulness of the results presented here since, assuming the utilized tool is sound, validation of a formal verification artifact merely requires validation of the formalized specification and related claims (as long as the claims are successfully verifiable in the tool). Nevertheless, all results have successfully been formally verified. The code corresponding to this formal verification is provided in the repository belonging to this work; this repository can be found at https://github.com/MM45/Saber-Formal-Verification-EasyCrypt.

Fundamental Specification. Foremost, we formalize the most rudimentary part of the considered context: Saber's parameters and the corresponding constraints. These formalizations are presented in Listing 1.1 and Listing 1.2, respectively.

```
const eq, ep, et: int.
const en: int.

const q: int = 2^eq.
const p: int = 2^ep.
const t: int = 2^et.
const n: int = 2^en.
const l: int.
```

Listing 1.1. Saber's Parameters

```
axiom zero_en: 0 <= en.
axiom one_et1: 1 <= et + 1.
axiom et2_ep: et + 2 <= ep.
axiom ep1_eq: ep + 1 <= eq.
axiom one_l: 1 <= l.
```

Listing 1.2. Parameter Constraints

Naturally, each constant defined in the former listing represents the similarly-named parameter of Saber; furthermore, the second, third, and fourth axioms in the latter listing together formalize $1 \leq \epsilon_t + 1 \wedge \epsilon_t + 2 \leq \epsilon_p \wedge \epsilon_p + 1 \leq \epsilon_q$, which is equivalent to the previously mentioned constraint that Saber enforces on these exponents, i.e., $0 < \epsilon_t + 1 < \epsilon_p < \epsilon_q$.

Subsequent to the parameters, we define the necessary types and operators[7]. First, most of the types we define are used to denote the algebraic structures employed in Saber; for example, we define `Zq`, `Rq`, `Rq_vec`, and `Rq_mat` to respectively denote $\mathbb{Z}_q$, R_q, $R_q^{l\times 1}$, and $R_q^{l\times l}$. The types for the remainder of the algebraic structures follow a similar identifier format, e.g., `Rp` represents R_p. Naturally, for each of these types, the appropriate structure is formalized and assigned. Second, the formalizations of the functions predominantly consist of operators that carry out modular reduction or modular scaling and flooring. Listing 1.3 contains the definitions for a selected few of these operators.

```
op Zq2Zp (z : Zq) : Zp = Zp.inzmod (Zq.asint z).

op scaleZq2Zp (z : Zq) : Zp =
        Zp.inzmod (shr (Zq.asint z) (eq - ep)).
op scaleZp2Zq (z : Zp) : Zq =
        Zq.inzmod (shl (Zp.asint z) (eq - ep)).
```

Listing 1.3. Modular Reduction and Modular Scaling and Flooring

Here, the `asint` operator converts from the associated integer ring type, e.g., `Zp` or `Zq`, to the integers; the `inzmod` operator performs the opposite conversion. In its conversion, `inzmod` implicitly reduces the provided argument modulo the corresponding modulus. Moreover, `shr` and `shl` respectively compute a right and left bit-shift of their first (integer) argument by a number of bits equal to their second (integer) argument. Apart from the specific operators shown in this

[7] In EasyCrypt, an operator denotes a mathematical function.

listing, we define variants for each type combination involved in a modular reduction or modular scaling and flooring throughout; this includes the extensions of these operators to (vectors of) polynomials. For intelligibility purposes, all of these operators share the same self-explanatory identifier format. Lastly, 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 other properties or requirements.

At this point, the only remaining fundamental artifacts to formalize are the required distributions. For the uniform distributions, we utilize the built-in mechanisms of EasyCrypt to construct precise formalizations; for type `X`, these distributions are denoted by `dX`. Contrariwise, instead of $\beta_\mu(R_q^{l \times 1})$, we formalize an arbitrary, generic distribution over $R_q^{l \times 1}$ that is denoted by `dsmallRq_vec`. Indeed, this generic distribution most definitely encompasses $\beta_\mu(R_q^{l \times 1})$. In consequence of this more abstract approach, the formal verification gives slightly stronger guarantees without requiring additional nontrivial assumptions; specifically, with this approach, the formal verification shows that the security proof is valid for any distribution (over $R_q^{l \times 1}$) in place of $\beta_\mu(R_q^{l \times 1})$, provided the MLWR game is hard with this distribution. Moreover, this approach precludes the (somewhat tedious) effort of precisely formalizing $\beta_\mu(R_q^{l \times 1})$; this is also the reason we do not formalize the μ parameter (see Listing 1.1).

Leveraging the above fundamentals, we can formalize the remainder of the necessary higher-level artifacts such as the specification of Saber.PKE, the security notion, the hardness assumptions, the game sequence, and the justifications for the steps in the game-based proof. Although we do not explicitly present every utilized artifact here, the ensuing discussion will cover several of them at the relevant places.

Step 1: $\text{Game}_{\mathcal{A}}^{0}$ - $\text{Game}_{\mathcal{A}}^{1}$. Before the formal verification of each step, and so also preceding the formal verification of the first step, we formalize the adversary class(es), games, and reduction adversary relevant to this step. Concerning the former, Listing 1.4 depicts the formalizations of the classes of IND-CPA adversaries and GMLWR adversaries, both utilized in the first step.

```
1  module type Adv_INDCPA = {
2    proc choose(pk : seed * Rp_vec) : R2 * R2
3    proc guess(c : R2t * Rp_vec) : bool
4  }.
5
6  module type Adv_GMLWR = {
7    proc guess(sd : seed, b : Rp_vec) : bool
8  }.
```

Listing 1.4. Classes of IND-CPA Adversaries and GMLWR Adversaries

A `module type` defines an abstract interface that modules may implement. EasyCrypt allows universal quantification over these types, enabling one to abstractly reason about every potential instantiation; therefore, this mechanism is well-

suited to capture the concept of an adversary class[8]. Regarding the module types presented above, `Adv_INDCPA` denotes the class of IND-CPA adversaries, where `choose` and `guess` respectively represent $\mathcal{A}.\mathsf{P}$ and $\mathcal{A}.\mathsf{D}$; `Adv_GMLWR` denotes the class of GMLWR adversaries, where `guess` represents the only algorithm of these adversaries. Naturally, the parameters and return values of these procedures accordingly formalize the parameters and return values of the corresponding algorithms.

Employing the formalization of the adversary classes, we formalize the necessary games. First, Listing 1.5 provides the formalization of $\mathrm{Game}^0_{\mathcal{A}}$. Here, as well as in all ensuing listings, the `(*...*)` comment represents the (uninteresting) omitted section of variable declarations.

```
module Game0(A : Adv_INDCPA) = {
   proc main() : bool = {
      (*...*)
      u <$ dbool;
      sd <$ dseed;
      _A <- gen sd;
      s <$ dsmallRq_vec;
      b <- scaleRqv2Rpv (_A *^ s + h);

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

      s' <$ dsmallRq_vec;
      b' <- scaleRqv2Rpv ((trmx _A) *^ s' + h);
      v' <- (dotp b (Rqv2Rpv s')) + (Rq2Rp h1);
      chat <- scaleRp2R2t (v' + (scaleR22Rp (
                if u then m1 else m0)));

      u' <@ A.guess((chat, b'));
      return (u = u');
   }
}.
```

Listing 1.5. $\mathrm{Game}^0_{\mathcal{A}}$

As exemplified in this listing, we formalize games through parameterized modules; specifically, the parameter provided to such modules is another module that formalizes the considered adversary. Moreover, the actual statements executed by the game are encapsulated in a procedure; however, this is merely a syntactical requirement of EasyCrypt. Statements in module procedures belong to one of several categories: regular assignment statements (using `<-`), sample statements (using `<$`), and procedure call statements (using `<@`). Then, noting that `*^`, `dotp`, `trmx`, `h1`, and `h` respectively denote matrix-vector multiplication, dot

[8] Nevertheless, albeit customary in hand-written cryptographic proofs, EasyCrypt currently does not provide the possibility to restrict the space or time complexity of module types.

product, transpose, h_1, and $\mathbf{h}$, we can see that `Game0(A).main()` is a line-by-line verbatim translation of $\text{Game}^0_{\mathcal{A}}$, where `A` formalizes $\mathcal{A}$. Furthermore, from this formalization of $\text{Game}^0_{\mathcal{A}}$, we can straightforwardly derive the formalization of $\text{Game}^1_{\mathcal{A}}$; indeed, we merely remove line 7 and replace line 8 with the appropriate sample statement.

The last game relevant to the first step is $\text{Game}^{\text{GMLWR}}_{\mathcal{A},\mathsf{gen},l,\mu,q,p}(u)$; the formalization of this game is presented in Listing 1.6.

```
1  module GMLWR(A : Adv_GMLWR) = {
2     proc main(u : bool) : bool = {
3        (*...*)
4        sd <$ dseed;
5        _A <- gen sd;
6        s <$ dsmallRq_vec;
7
8        if (u) {
9           b <$ dRp_vec;
10       } else {
11          b <- scaleroundRqv2Rpv (_A *^ s);
12       }
13       u' <@ A.guess(sd, b);
14       return u';
15    }
16 }.
```

Listing 1.6. $\text{Game}^{\text{GMLWR}}_{\mathcal{A},\mathsf{gen},l,\mu,q,p}(u)$

Here, `scaleroundRqv2Rpv` formalizes the extension (to polynomial vectors) of the modular scaling and rounding function employed in $\text{Game}^{\text{GMLWR}}_{\mathcal{A},\mathsf{gen},l,\mu,q,p}(u)$. Taking this into account, we can see that `GMLWR(A).main(u)` is a correct formalization of $\text{Game}^{\text{GMLWR}}_{\mathcal{A},\mathsf{gen},l,\mu,q,p}(u)$, where `A` and `u` respectively denote $\mathcal{A}$ and u[9].

Penultimately, we formalize adversary $\mathcal{B}_0^{\mathcal{A}}$ against $\text{Game}^{\text{GMLWR}}_{\mathcal{B}_0^{\mathcal{A}},\mathsf{gen},l,\mu,q,p}(u)$. More specifically, we formalize this reduction adversary as a parameterized module `B0(A : Adv_INDCPA)` of type `Adv_GMLWR`. Indeed, the `Adv_GMLWR` type enforces `B0(A)` to implement `guess(sd, b)`; in this case, this procedure precisely formalizes the $\mathcal{B}_0^{\mathcal{A}}(\text{seed}_{\mathbf{A}}, \mathbf{b}_u)$ presented in Figure 6. Due to the similarities between $\mathcal{B}_0^{\mathcal{A}}$ and the initial two games in the game sequence, `B0(A).guess(sd, b)` is nearly identical to the above-discussed formalizations of $\text{Game}^0_{\mathcal{A}}$ and $\text{Game}^1_{\mathcal{A}}$. For this reason, we refrain from explicitly presenting the formalization of this reduction adversary here.

Finally, we formalize the result of the security proof's first step. In particular, we do so by formulating an appropriate lemma; Listing 1.7 provides this lemma.

[9] Remark that the considered parameters of $\text{Game}^{\text{GMLWR}}_{\mathcal{A},\mathsf{gen},l,\mu,q,p}(u)$ are the same as the similarly named (function and) parameters of Saber; hence, these are already formalized outside of the `GMLWR` module, see the foregoing discussion concerning the fundamental specification.

```
1  lemma Step_Distinguish_Game0_Game1_GMLWR &m :
2    `| Pr[Game0(A).main() @ &m : res] -
3       Pr[Game1(A).main() @ &m : res] |
4    =
5    `| Pr[GMLWR( B0(A) ).main(true) @ &m : res] -
6       Pr[GMLWR( B0(A) ).main(false) @ &m : res] |.
```

Listing 1.7. First Step in Game-Playing Security Proof

In this lemma, `A` is an arbitrary module of type `Adv_INDCPA`; that is, `A` formalizes an arbitrary IND-CPA adversary. Furthermore, `&m` signifies an arbitrary memory that, in this case, essentially formalizes the context in which the games and adversaries are executed. Then, given that `` `|x| ``, `Pr[E]`, and `res` respectively denote the absolute value of `x`, probability of `E`, and return value of the considered procedure, we can recognize the following correspondences.

$$\texttt{Pr[Game0(A).main() @ \&m : res]} \cong \Pr\left[\mathrm{Game}_{\mathcal{A}}^{0} = 1\right]$$

$$\texttt{Pr[Game1(A).main() @ \&m : res]} \cong \Pr\left[\mathrm{Game}_{\mathcal{A}}^{1} = 1\right]$$

$$\texttt{Pr[GMLWR(B0(A)).main(true) @ \&m : res]} \cong \Pr\left[\mathrm{Game}^{\mathrm{GMLWR}}_{\mathcal{B}_0^{\mathcal{A}},\mathsf{gen},l,\mu,q,p}(1) = 1\right]$$

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

This demonstrates that `Step_Distinguish_Game0_Game1_GMLWR` correctly formalizes the result of the initial step of the security proof.

Step 2: $\mathrm{Game}_{\mathcal{A}}^{1}$ - $\mathrm{Game}_{\mathcal{A}}^{2}$. For the formal verification of the second step, similarly to before, we first formalize the required games and reduction adversary. Concerning the games, since $\mathrm{Game}_{\mathcal{A}}^{1}$ has already been formalized in the preceding step, we only need to formalize $\mathrm{Game}_{\mathcal{A}}^{2}$. The formalization of $\mathrm{Game}_{\mathcal{A}}^{2}$ can straightforwardly be derived from the formalization of $\mathrm{Game}_{\mathcal{A}}^{0}$ presented in Listing 1.5; more precisely, this can be achieved by removing line 7, changing line 8 to the proper sampling statement, and appropriately modifying the computation of `chat` in line 15. Regarding the reduction adversary, we construct a module `A2(A1 : Adv_INDCPA)` of type `Adv_INDCPA_2`. This module type is identical to `Adv_INDCPA`, except for the fact that the parameter of its `guess` procedure is of type `Rppq * Rp_vec` instead of type `R2t * Rp_vec`; this models that the ciphertext given to the adversary in $\mathrm{Game}_{\mathcal{A}}^{2}$ is an element of $R_{p^2/q} \times R_p^{l\times 1}$ instead of $R_{2\cdot t} \times R_p^{l\times 1}$. In other words, while `Adv_INDCPA` formalizes the class of adversaries against $\mathrm{Game}^{\text{IND-CPA}}_{\mathcal{A},\text{Saber.PKE}}$, $\mathrm{Game}_{\mathcal{A}}^{0}$, and $\mathrm{Game}_{\mathcal{A}}^{1}$, `Adv_INDCPA_2` formalizes the class of adversaries against $\mathrm{Game}_{\mathcal{A}}^{2}$[10]. In accordance with its module type, `A2` implements `choose(pk)` and `guess(c)`, respectively formalizing $\mathcal{R}^{\mathcal{A}}.\mathsf{P}((\mathrm{seed}_{\mathbf{A}}, \mathbf{b}))$ and $\mathcal{R}^{\mathcal{A}}.\mathsf{D}((\hat{c}, \mathbf{b}'))$ as specified in Figure 7. Indeed, `A2(A1).choose(pk)` merely returns `A1.choose(pk)`; `A2(A1).guess(c)` performs an appropriate modular scaling

[10] Analogously, there is a separate module type that formalizes the class of adversaries against $\mathrm{Game}_{\mathcal{A}}^{3}$ and $\mathrm{Game}_{\mathcal{A}}^{4}$.

and flooring operation on the first element of `c` before returning `A1.guess(c')`, where `c'` denotes the adjusted `c`.

Having constructed the necessary formalizations, we can express the lemma that formalizes the result of the security proof's second step; this lemma is presented in Listing 1.8.

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

Listing 1.8. Second Step in Game-Playing Security Proof

In this lemma, as in `Step_Distinguish_Game0_Game1_GMLWR`, `A` denotes an arbitrary module of type `Adv_INDCPA`, i.e., the formalization of an arbitrary IND-CPA adversary. Then, given the previously explained interpretation of the employed statements and the fact that `1%r / 2%r` denotes $\frac{1}{2}$, we can see that `Step_Game1_Game2` accurately formalizes the result of the second step of the security proof.

Step 3, Step 4, and Analysis of $\mathrm{Game}^4_{\mathcal{A}}$. Based on the preceding discussion, the formal verification process for the remaining two steps of the security proof can straightforwardly be extrapolated. Namely, the formal verification of the third step, reducing from $\mathrm{Game}^3_{\mathcal{A}}$ to $\mathrm{Game}^2_{\mathcal{A}}$, follows an analogous procedure to the second step; the formal verification of the fourth step, reducing from $\mathrm{Game}^{\mathrm{XMLWR}}_{\mathcal{A},\mathsf{gen},l,\mu,q,p}(u)$ to distinguishing between $\mathrm{Game}^3_{\mathcal{A}}$ and $\mathrm{Game}^4_{\mathcal{A}}$, has a similar structure to the first step.

Regarding the formal verification of the $\frac{1}{2}$ winning probability of any adversary against $\mathrm{Game}^4_{\mathcal{A}}$, we foremost formally verify the equivalence between $\mathrm{Game}^4_{\mathcal{A}}$ and a contrived auxiliary game; this auxiliary game is identical to $\mathrm{Game}^4_{\mathcal{A}}$, except that it samples every artifact from the appropriate uniform distribution and delays the sampling of u to the final statement (preceding the return statement). Due to its construction, contrary to $\mathrm{Game}^4_{\mathcal{A}}$, the auxiliary game facilitates the formal verification of the invariable $\frac{1}{2}$ winning probability. Afterward, the fact that any adversary against $\mathrm{Game}^4_{\mathcal{A}}$ has a $\frac{1}{2}$ winning probability directly follows from the equivalence between $\mathrm{Game}^4_{\mathcal{A}}$ and the auxiliary game.

Final Result (Security Theorem). Finally, we consider the security theorem. Similarly to the security proof's steps, this theorem is formalized by formulating a suitable lemma; this lemma is provided in Listing 1.9.

```
lemma Saber_INDCPA_Security_Theorem &m :
  exists (BG <: Adv_GMLWR) (BX <: Adv_XMLWR),
  `| Pr[CPA(Saber_PKE_Scheme, A).main() @ &m : res] -
     1%r / 2%r |
  <=
  `| Pr[GMLWR(BG).main(true) @ &m : res] -
     Pr[GMLWR(BG).main(false) @ &m : res] |
  +
  `| Pr[XMLWR(BX).main(true) @ &m : res] -
     Pr[XMLWR(BX).main(false) @ &m : res] |.
```

Listing 1.9. Security Theorem

Once again, `A` denotes an arbitrary module of type `Adv_INDCPA`. Furthermore, `CPA` and `XMLWR` are modules respectively comprising the formalizations of the IND-CPA and XMLWR games. In contrast to the modules considered hitherto, the `CPA` module is provided by EasyCrypt's standard library and defined with respect to a generic PKE scheme. For this reason, the `CPA` module needs to be instantiated with the desired concrete PKE scheme via its first parameter; naturally, in this case we use the module that formalizes Saber.PKE, `Saber_PKE_Scheme`, as the concrete PKE scheme. Lastly, `exists (BG <: Adv_GMLWR) (BX <: Adv_XMLWR)` constitutes an existential quantifier over a module of type `Adv_GMLWR` and a module of type `Adv_XMLWR`. Combining this with the foregoing material, we can see that `Saber_INDCPA_Security_Theorem` exactly formalizes the security theorem, as desired. This completes the formal verification of Saber.PKE's IND-CPA security property.

4 Correctness

Proceeding from the discussion on Saber.PKE's security property, we now consider the scheme's correctness property. In particular, we do so by discussing the most important parts of the devised hand-written proof and corresponding formal verification, akin to the foregoing discussion on the security property.

Alternative Specification and Correctness Notion. Before advancing to the actual hand-written proof of Saber.PKE's correctness property, we establish an alternative, yet equivalent, specification of Saber.PKE[11]; additionally, we introduce the relevant notion of correctness.

Foremost, to refer to the alternative specification of Saber.PKE, we use Saber.PKEA; furthermore, the key generation, encryption, and decryption algorithms of Saber.PKEA are respectively denoted by Saber.KeyGenA, Saber.EncA, and Saber.DecA. For the latter two algorithms, Algorithm 4 and Algorithm 5 provide the corresponding specifications; the specification of Saber.KeyGenA is identical to that of Saber.KeyGen and, therefore, not explicitly presented here.

[11] This alternative specification is based on the alternative specification of Saber's key exchange scheme presented in [D'A21].

Algorithm 4 Alternative Specification of Saber's Encryption Algorithm

procedure Saber.EncA(pk := ($\text{seed}_{\mathbf{A}}, \mathbf{b}$), m)
  $\mathbf{A} \leftarrow \mathsf{gen}(\text{seed}_{\mathbf{A}})$
  $\mathbf{s}' \leftarrow\$ \beta_\mu(R_q^{l\times 1})$
  $\mathbf{b}' \leftarrow \lfloor \mathbf{A}^T \cdot \mathbf{s}' + \mathbf{h} \rfloor_{q\to p}$
  $\mathbf{b}_q \leftarrow \lfloor \mathbf{b} \rfloor_{p\to q}$
  $v' \leftarrow \mathbf{b}_q^T \cdot \mathbf{s}' + \frac{q}{p} \cdot h_1$
  $c_m \leftarrow \lfloor v' + \lfloor m \rfloor_{2\to q} \rfloor_{q\to 2\cdot t}$
  return $c := (c_m, \mathbf{b}')$

Algorithm 5 Alternative Specification of Saber's Decryption Algorithm

procedure Saber.DecA(sk := $\mathbf{s}$, $c := (c_m, \mathbf{b}')$)
  $\mathbf{b}'_q \leftarrow \lfloor \mathbf{b}' \rfloor_{p\to q}$
  $v \leftarrow \mathbf{b}_q'^T \cdot \mathbf{s} + \frac{q}{p} \cdot h_1$
  $m' \leftarrow \lfloor v - \lfloor c_m \rfloor_{2\cdot t\to q} + \frac{q}{p} \cdot h_2 \rfloor_{q\to 2}$
  return m'

As shown in Algorithm 4 and Algorithm 5, Saber.EncA and Saber.DecA ensure that all of their operations exclusively involve elements from R_q (or $R_q^{l\times 1}$). This is accomplished by carrying out the appropriate modular scaling and flooring operations on elements that do not originate from these structures. For instance, Saber.EncA performs $\lfloor \mathbf{b} \rfloor_{p\to q}$ and $\lfloor m \rfloor_{2\to q}$; similarly, Saber.DecA performs $\lfloor \mathbf{b}' \rfloor_{p\to q}$ and $\lfloor c_m \rfloor_{2\cdot t\to q}$[12]. Furthermore, to guarantee their equivalence to the original specifications despite these differences, Saber.EncA and Saber.DecA consistently multiply h_1 and h_2 by $\frac{q}{p} \in R_q$. These alternative encryption and decryption algorithms can intuitively be seen to be equivalent to their original counterparts by noting that Saber.EncA and Saber.DecA essentially perform the same operations as Saber.Enc and Saber.Dec, except that certain elements considered in these operations contain additional appended zero bits. The primary rationale behind adopting this alternative specification for the correctness analysis is that it provides a convenient way to describe the errors induced by some of the modular scaling and flooring operations as elements from R_q (or $R_q^{l\times 1}$); this especially simplifies the corresponding formal verification by minimizing the number of different types considered throughout.

Regarding the correctness notion, as aforementioned, we employ the definition provided in [HHK17]. For convenience, we restate this definition for a generic PKE scheme here[13]; specifically, using the game presented in Figure 10,

[12] Recall that $0 < \epsilon_t + 1 < \epsilon_p < \epsilon_q$ and, hence, $2 \leq 2 \cdot t < p < q$; as a consequence, $\lfloor \cdot \rfloor_{2\to q}$, $\lfloor \cdot \rfloor_{2\cdot t\to q}$, and $\lfloor \cdot \rfloor_{p\to q}$ effectively constitute left bit-shifts.

[13] Actually, the definition we present and use is slightly different from, yet trivially equivalent to, the definition provided in [HHK17]. Namely, the definition we utilize considers the success probability instead of the failure probability; this is the only difference with the definition from [HHK17].

we say PKE is δ-correct if for all $\mathcal{A}$, the following holds.

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

In the concrete cases of Saber.PKE and Saber.PKEA, $\delta \neq 0$ due to the errors caused by several of the modular scaling and flooring operations; moreover, from the equivalence between Saber.PKE and Saber.PKEA, it follows that $\Pr\left[\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{Saber.PKEA}} = 1\right]$ is equal to $\Pr\left[\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{Saber.PKE}} = 1\right]$ for any $\mathcal{A}$.

$\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{PKE}}$

1 : $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}()$
2 : $m \leftarrow \mathcal{A}(\mathsf{pk}, \mathsf{sk})$
3 : $c \leftarrow \mathsf{Enc}(\mathsf{pk}, m)$
4 : $m' \leftarrow \mathsf{Dec}(\mathsf{sk}, c)$
5 : **return** $(m' = m)$

Figure 10. Correctness Game

$\text{PProg}^{\delta\text{COR}}$

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** ccrng(err_expression$(\mathbf{A}, \mathbf{s}, \mathbf{s}')$)

Figure 11. Probabilistic Program For Correctness Based on Error Expression

Hand-Written Proof. As alluded to above, Saber.PKEA and, by equivalence, Saber.PKE are not perfectly correct; that is, $\Pr\left[\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{Saber.PKEA}} = 1\right] < 1$. Concerning the specification of $\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{Saber.PKEA}}$, we can see that this game essentially verifies whether m' equals m after the sequential execution of $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Saber.KeyGenA}()$, $c \leftarrow \mathsf{Saber.EncA}(\mathsf{pk}, m)$, and $m' \leftarrow \mathsf{Saber.DecA}(\mathsf{sk}, c)$. 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. 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. Ultimately, as the remainder will show, the expression derived for the verification of $m' = m$, henceforth referred to as "error expression", exclusively depends on these error terms[14]. In turn, because these error terms, when fully expanded, only depend on randomly sampled artifacts (and constants) from the algorithms of Saber.PKEA, one can exhaustively compute the distribution of the error expression and, hence, the probability that this expression lies within a certain range. In fact, the authors of Saber have constructed a script that performs this exact computation, (indirectly) claiming that this is equivalent to computing the correctness of Saber.PKE (independent of the message) [D'A21,DKRV18]. Therefore, the ensuing proof aims to show that this probability computation indeed computes the δ such that $\Pr\left[\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{PKE}} = 1\right] = 1 - \delta$ holds for any $\mathcal{A}$.

[14] Similarly to the specification of Saber.PKEA, the definitions of these error terms are inspired by the error terms provided in [D'A21].

The first error term we define relates to $\mathbf{A} \cdot \mathbf{s}$ and $\mathbf{b}_q$; particularly, 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.

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

The second error term is similar 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 $\mathbf{b}_q$. The definition of this error term is given below.

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

The final error term captures the error related to $v' + \lfloor m \rfloor_{2 \to q}$ and $\lfloor c_m \rfloor_{2 \cdot t \to q}$; this error term is defined below.

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

In contrast to the other error terms, $\mathrm{err}_{c_{mq}}$ adds a constant that is not present in the related modular scaling and flooring operations. This constant, i.e., $\frac{q}{4 \cdot t} \in R_q$, ensures that the coefficients of $\mathrm{err}_{c_{mq}}$ are centered around zero; indeed, the coefficients of $\mathbf{err}_{\mathbf{b}_q}$ and $\mathbf{err}_{\mathbf{b}'_q}$ are already centered around zero as is.

Utilizing the above-introduced error terms, we presently derive the error expression. To this end, considering the sequential execution of Saber.PKEA's algorithms, we first rewrite $v - \lfloor c_m \rfloor_{2 \cdot t \to q} + \frac{q}{p} \cdot h_2$ as follows. At times, to prevent cluttering, we replace $\lfloor m \rfloor_{2 \to q} + \frac{q}{4 \cdot t} + \frac{q}{p} \cdot h_2$ by horizontal dots.

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

In this derivation, most equalities follow from trivial substitutions, reorderings, simplifications, and basic operator properties; however, confirming the validity of the fifth and final equality might require some more thought. Specifically, the fifth equality holds due to the fact that $\mathbf{s}'^T \cdot \mathbf{A} \cdot \mathbf{s}$ and $\mathbf{s}^T \cdot \mathbf{A}^T \cdot \mathbf{s}'$ are equal and, as such, cancel each other out. Observing that both of these terms are elements

of R_q, i.e., no vectors[15], this can be deduced as shown below.

$$\mathbf{s'}^T \cdot \mathbf{A} \cdot \mathbf{s} = ((\mathbf{s'}^T \cdot \mathbf{A} \cdot \mathbf{s})^T)^T = (\mathbf{s}^T \cdot \mathbf{A}^T \cdot \mathbf{s'})^T = \mathbf{s}^T \cdot \mathbf{A}^T \cdot \mathbf{s'}$$

The final equality of the preceding derivation is valid because $\lfloor m \rfloor_{2\to q}$ is equal to $-\lfloor m \rfloor_{2\to q}$. Particularly, since each coefficient of m equals either 0 or 1, each coefficient of its scaled counterpart $\lfloor m \rfloor_{2\to q}$ has value $0 \cdot \frac{q}{2} = 0$ or $1 \cdot \frac{q}{2} = \frac{q}{2}$; consequently, each coefficient of $-\lfloor m \rfloor_{2\to q}$ equals $-0 = 0$ or $-\frac{q}{2}$. Naturally, in R_q, these corresponding coefficients of $\lfloor m \rfloor_{2\to q}$ and $-\lfloor m \rfloor_{2\to q}$ are equivalent.

Additionally considering the modular scaling and flooring operation that Saber.DecA applies to $v - \lfloor c_m \rfloor_{2^{\cdot t}\to q} + \frac{q}{p} \cdot h_2$, we can utilize the above-derived expression to determine the final value Saber.DecA assigns to m'.

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

Here, the last equality can be seen to hold by considering that m is an element of R_2, $\frac{q}{4} + \mathbf{err}_{\mathbf{b'}_q}^T \cdot \mathbf{s} - \mathbf{err}_{\mathbf{b}_q}^T \cdot \mathbf{s'} - \mathrm{err}_{c_{mq}}$ is an element of R_q, and $\lfloor \cdot \rfloor_{2\to q}$ and $\lfloor \cdot \rfloor_{q\to 2}$ respectively perform left and right bit-shifts of $\epsilon_q - 1$ bits. Namely, this implies that both sides of the last equality add the (single) bit of each coefficient of m to the most significant bit of the corresponding coefficient of $\frac{q}{4} + \mathbf{err}_{\mathbf{b'}_q}^T \cdot \mathbf{s} - \mathbf{err}_{\mathbf{b}_q}^T \cdot \mathbf{s'} - \mathrm{err}_{c_{mq}}$.

At this point, it is rather trivial to derive that $m' = m$ if and only if $\lfloor \frac{q}{4} + \mathbf{err}_{\mathbf{b'}_q}^T \cdot \mathbf{s} - \mathbf{err}_{\mathbf{b}_q}^T \cdot \mathbf{s'} - \mathrm{err}_{c_{mq}} \rfloor_{q\to 2} = 0$. In turn, the latter is veracious if and only if each coefficient of $\frac{q}{4} + \mathbf{err}_{\mathbf{b'}_q}^T \cdot \mathbf{s} - \mathbf{err}_{\mathbf{b}_q}^T \cdot \mathbf{s'} - \mathrm{err}_{c_{mq}}$ lies in the (discrete) range $[0, \frac{q}{2})$. Finally, subtracting the $\frac{q}{4}$ constant, we obtain the desired result: $m' = m$ 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 $[-\frac{q}{4}, \frac{q}{4})$. Indeed, $\mathbf{err}_{\mathbf{b'}_q}^T \cdot \mathbf{s} - \mathbf{err}_{\mathbf{b}_q}^T \cdot \mathbf{s'} - \mathrm{err}_{c_{mq}}$ constitutes the desired error expression.

As a last endeavor preceding the conclusion of this proof, we show 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. To this end, consider the part of the error term that includes m, i.e., $\lfloor \lfloor v' + \lfloor m \rfloor_{2\to q} \rfloor_{q\to 2^{\cdot t}} \rfloor_{2^{\cdot t}\to q} - (v' + \lfloor m \rfloor_{2\to q})$. Akin to before, interpreting the modular scaling and flooring operations as bit-shifts allows to deduce that $\lfloor \lfloor v' + \lfloor m \rfloor_{2\to q} \rfloor_{q\to 2^{\cdot t}} \rfloor_{2^{\cdot t}\to q}$ is identical to $v' + \lfloor m \rfloor_{2\to q}$ in its $\epsilon_q - \epsilon_t - 1$ most significant bits, but exclusively contains zero bits otherwise. Since $\lfloor m \rfloor_{2\to q}$ can only affect the value of the most significant (ϵ_q-th) bit of $v' + \lfloor m \rfloor_{2\to q}$, it follows that $\lfloor \lfloor v' + \lfloor m \rfloor_{2\to q} \rfloor_{q\to 2^{\cdot t}} \rfloor_{2^{\cdot t}\to q}$ is equal to $\lfloor m \rfloor_{2\to q} + \lfloor \lfloor v' \rfloor_{q\to 2^{\cdot t}} \rfloor_{q\to 2^{\cdot t}}$. Consequently, we can rewrite $\mathrm{err}_{c_{mq}}$ as shown below, demonstrating the error

[15] Remark that this implies the transpose reduces to the identity function.

term's independence of m.

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

Finally, utilizing the obtained results, we can infer that for any $\mathcal{A}$, computing $\Pr\left[\mathrm{Game}^{\mathrm{FOCOR}}_{\mathcal{A},\mathrm{Saber.PKEA}} = 1\right]$ is equivalent to computing the probability that all coefficients of the corresponding error expression lie in the discrete range $[-\frac{q}{4}, \frac{q}{4})$. Furthermore, because this error expression is independent of the message, this probability does not depend on the particular m. In fact, completely unfolding the error expression, we can see that excluding any constants, it solely depends on $\mathbf{A}$, $\mathbf{s}$, and $\mathbf{s}'$ produced as in Saber.KeyGenA, Saber.KeyGenA, and Saber.EncA, respectively. As such, assuming gen's output distribution is uniformly random, we can formalize the probability computation based on the error expression as the probabilistic program defined in Figure 11; this precisely denotes the probability computation performed by the aforementioned script constructed by Saber's authors[16] [D'A21]. Here, err_expression$(\mathbf{A}, \mathbf{s}, \mathbf{s}')$ represents the error expression $\mathbf{err}^T_{\mathbf{b}'_q} \cdot \mathbf{s} - \mathbf{err}^T_{\mathbf{b}_q} \cdot \mathbf{s}' - \mathrm{err}_{c_{mq}}$, accordingly using the provided arguments as the values for $\mathbf{A}$, $\mathbf{s}$, and $\mathbf{s}'$; moreover, for $x \in R_q$, ccrng(x) denotes the predicate that evaluates to true if and only if each of x's coefficients lies in $[-\frac{q}{4}, \frac{q}{4})$.

Using $\mathrm{PProg}^{\delta\mathrm{COR}}$ and the equivalence between Saber.PKE and Saber.PKEA, we can derive the following concluding sequence of equalities, assuming the uniformity of gen's output distribution; certainly, these equalities hold for any adversary $\mathcal{A}$.

$$\Pr\left[\mathrm{Game}^{\mathrm{FOCOR}}_{\mathcal{A},\mathrm{Saber.PKE}} = 1\right] = \Pr\left[\mathrm{Game}^{\mathrm{FOCOR}}_{\mathcal{A},\mathrm{Saber.PKEA}} = 1\right] = \Pr\left[\mathrm{PProg}^{\delta\mathrm{COR}} = 1\right]$$

Formal Verification. Having discussed the hand-written correctness proof of Saber.PKE, we imminently cover the essential parts of the corresponding formal verification in EasyCrypt. More precisely, in the ensuing, we address the formalization of the error expression, $\mathrm{Game}^{\mathrm{FOCOR}}_{\mathcal{A},\mathrm{PKE}}$, and $\mathrm{PProg}^{\delta\mathrm{COR}}$; furthermore, we consider the formal verification of the relevant program equivalences and the final desired equality of probabilities. As with the discussion on the formal verification of Saber.PKE's security property (and for the same reasons), we do not go over the concrete proofs of the results in EasyCrypt; however, all results have successfully been formally verified. Again, the code corresponding to this formal verification is provided in the repository belonging to this work.

[16] This suggests that the script merely approximates the actual correctness value. Nevertheless, if gen is adequately instantiated, i.e., its output distribution (closely) resembles the uniform distribution, this approximation is (almost) accurate.

Starting off, we formalize $\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{PKE}}$, i.e., the correctness game regarding a generic PKE scheme. Similarly to the formalization of the games in the security proof, the formalization of this correctness game requires the formalization of the considered adversary class; the latter is given in Listing 1.10.

```
module type Adv_Cor = {
   proc choose(pk : seed * Rp_vec, sk : Rq_vec) : R2
}.
```

Listing 1.10. Class of FOCOR Adversaries

The interpretation of `Adv_Cor` and `choose` is trivially extrapolated from the discussion surrounding Listing 1.4.

Employing the above formalization of the considered adversary class, the formalization of $\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{PKE}}$ is provided in Listing 1.11.

```
module Cor_Game (S : Scheme, A : Adv_Cor) = {
   proc main() : bool = {
      (*...*)
      (pk, sk) <@ S.kg();
      m <@ A.choose(pk, sk);
      c <@ S.enc(pk, m);
      m' <@ S.dec(sk, c);
      return (m' = Some m);
   }
}.
```

Listing 1.11. $\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{PKE}}$

Modeling the fact that $\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{PKE}}$ is defined with respect to a generic PKE scheme, `Cor_Game` takes an additional parameter of the built-in (module) type `Scheme`. This module type is designed for the formalization of PKE schemes; as such, it defines the `kg`, `enc`, and `dec` procedures, respectively representing the key generation, encryption, and decryption algorithms of PKE schemes. Taking this into account, we can see that the definition of `Cor_Game` is a verbatim translation of $\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{PKE}}$ to EasyCrypt[17].

Leveraging `Cor_Game`, we formalize the equivalence between $\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{Saber.PKE}}$ and $\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{Saber.PKEA}}$ as shown in Listing 1.12. Here, as the identifiers suggest, `Saber_PKE_Scheme` formalizes Saber.PKE and `Saber_PKE_Scheme_Alt` formalizes Saber.PKEA. Furthermore, `A` constitutes an arbitrary module of type `Adv_Cor`; that is, `A` formalizes an arbitrary adversary from the class of adversaries against the correctness game. As such, `Cor_Game(Saber_PKE_Scheme, A).main` for-

[17] The `Some` in the return statement is a technical consequence of the fact that certain PKE schemes may explicitly indicate decryption failure; nevertheless, this is irrelevant to the current discussion and, thus, can be ignored. Alternatively stated, we can regard the return statement as being `return (m' = m)`

malizes $\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{Saber.PKE}}$ and `Cor_Game(Saber_PKE_Scheme_Alt, A).main` formalizes $\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{Saber.PKEA}}$, both for an arbitrary $\mathcal{A}$.

```
lemma Equivalence_Cor_Game_Orig_Alt :
  equiv[Cor_Game(Saber_PKE_Scheme, A).main ~
        Cor_Game(Saber_PKE_Scheme_Alt, A).main
        : ={glob A} ==> ={res}].
```

Listing 1.12. Equivalence Between $\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{Saber.PKE}}$ and $\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{Saber.PKEA}}$

Substantiating that `Equivalence_Cor_Game_Orig_Alt` accurately formalizes the desired equivalence, we elaborate on the interpretation of this lemma. Foremost, an `equiv` statement consists of two primary parts, both contained within the square brackets, separated by a colon: the part preceding the colon specifies the collated procedures (separated by `~`); the part succeeding the colon specifies the pre- and postconditions under which the equivalence holds (separated by `==>`). So, in the `equiv` statement above, `Cor_Game(Saber_PKE_Scheme, A).main` is collated with `Cor_Game(Saber_PKE_Scheme_Alt, A).main`. Moreover, `={glob A}` and `={res}` respectively constitute the pre- and postconditions under which the equivalence should hold. More precisely, the precondition, `={glob A}`, states that in the execution of both considered programs, the initial perspectives of `A` should be identical; the postcondition, `={res}`, denotes that for all possible output values, the probability that one of the programs returns this value equals the probability that the other program outputs this (same) value. Thus, essentially, `Equivalence_Cor_Game_Orig_Alt` formalizes that for all $\mathcal{A}$ against $\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{Saber.PKE}}$ and $\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{Saber.PKEA}}$, the probability of the former game outputting a certain value equals the probability of the latter game outputting that same value. This is sufficient for this formal verification.

Following, in order to formalize the equivalence between $\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{Saber.PKEA}}$ and $\text{PProg}^{\delta\text{COR}}$ (for all $\mathcal{A}$), we must first formalize $\text{PProg}^{\delta\text{COR}}$; in turn, this requires the formalization of, in particular, the error terms and error expression. Recall that the error terms are defined as follows, using the previously derived definition of $\text{err}_{c_{mq}}$ that does not include m.

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

Then, illustrating the manner in which these error terms are formalized, we examine the formalization of $\mathbf{err}_{\mathbf{b}_q}$; this formalization is provided in Listing 1.13.

```
op error_bq (_A : Rq_mat) (s : Rq_vec) : Rq_vec =
  (scaleRpv2Rqv (scaleRqv2Rpv (_A *^ s + h))) -
    (_A *^ s).
```

Listing 1.13. $\mathbf{err}_{\mathbf{b}_q}$

As this listing demonstrates, we formalize the error terms as parameterized operators. The parameters of these operators are intended to be instantiated with the appropriate artifacts generated by (the formalization of) $\text{PProg}^{\delta\text{COR}}$; if this is the case, the operators accurately formalize the error terms. For instance, the parameters `_A` and `s` of `error_bq` are expected to be instantiated with (the formalizations of) $\mathbf{A}$ and $\mathbf{s}$, respectively. If the parameters are instantiated as such, we can see that `(scaleRpv2Rqv (scaleRqv2Rpv (_A *^ s + h))) - (_A *^ s)` precisely formalizes $\lfloor\lfloor \mathbf{A} \cdot \mathbf{s} + \mathbf{h} \rfloor_{q \to p} \rfloor_{p \to q} - \mathbf{A} \cdot \mathbf{s}$; hence, with appropriate parameter instantiations, `error_bq` accurately formalizes $\mathbf{err}_{\mathbf{b}_q}$.

Next, we reiterate that, based on the error terms, the error expression is defined as given below.

$$\mathbf{err}_{\mathbf{b}'_q}^T \cdot \mathbf{s} - \mathbf{err}_{\mathbf{b}_q}^T \cdot \mathbf{s}' - \text{err}_{c_{mq}}$$

Utilizing the above-discussed formalizations of $\mathbf{err}_{\mathbf{b}_q}^T$, $\mathbf{err}_{\mathbf{b}'_q}^T$, and $\text{err}_{c_{mq}}$, formalizing the error expression is relatively easy. Nevertheless, for completeness, Listing 1.14 provides the resulting formalization.

```
op error_expression (_A : Rq_mat) (s s': Rq_vec) =
  dotp (error_bq' _A s') s - dotp (error_bq _A s) s' -
    error_cmq _A s s'
```

Listing 1.14. Error Expression

Employing `error_expression`, we can construct the formalization of $\text{PProg}^{\delta\text{COR}}$; this formalization is presented in Listing 1.15.

```
module Delta_Cor_PProg = {
   proc main() : bool = {
      (*...*)
      _A <$ dRq_mat;
      s <$ dsmallRq_vec;
      s' <$ dsmallRq_vec;
      return ccrng (error_expression _A s s');
   }
}.
```

Listing 1.15. $\text{PProg}^{\delta\text{COR}}$

As expected, since $\text{PProg}^{\delta\text{COR}}$ merely comprises three sampling operations and a return statement, the definition of `Delta_Cor_PProg` is rather straightforward. Specifically, the only novelty in this definition concerns the `ccrng` operator in the return statement. As its identifier suggests, this operator merely formalizes the ccrng predicate. That is, `ccrng` takes an argument of type `Rq` and evaluates to true if and only if all of the argument's coefficients lie between $-\frac{q}{4}$ (including) and $\frac{q}{4}$ (excluding).

Harnessing the formalizations of $\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{Saber.PKEA}}$ and $\text{PProg}^{\delta\text{COR}}$, we can formalize the equivalence between these games; Listing 1.16 provides the formalization of this equivalence.

```
lemma Equivalence_CorGame_DeltaCorPProg :
  equiv[Cor_Game(Saber_PKE_Scheme_Alt, A).main ~
        Delta_Cor_PProg.main : true ==> ={res}].
```

Listing 1.16. Equivalence Between $\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{Saber.PKEA}}$ and $\text{PProg}^{\delta\text{COR}}$

As in preceding correctness-related listings, `A` constitutes an arbitrary module of type `Adv_Cor`, formalizing an arbitrary $\mathcal{A}$ against $\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{Saber.PKEA}}$. Furthermore, the interpretation of `Equivalence_CorGame_DeltaCorPProg` is similar to the interpretation of the equivalence lemma provided in Listing 1.12. More precisely, `Equivalence_CorGame_DeltaCorPProg` formalizes that for all $\mathcal{A}$ against $\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{Saber.PKEA}}$, the probability that $\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{Saber.PKEA}}$ outputs a specific value is equal to the probability that $\text{PProg}^{\delta\text{COR}}$ outputs that same value; this holds for all possible output values.

Finally, albeit trivially veracious at this point, we formally verify the fact that for any $\mathcal{A}$ against $\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{Saber.PKE}}$, $\Pr\left[\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{Saber.PKE}} = 1\right]$ is equal to $\Pr\left[\text{PProg}^{\delta\text{COR}} = 1\right]$. Listing 1.17 contains the formalization of this statement. Given that, once again, `A` is an arbitrary module of type `Adv_Cor`, the interpretation of the lemma in this listing should be evident from previous listings and corresponding discussions.

```
lemma Eq_Prob_CorGameOrig_DeltaCorPProg &m :
  Pr[Cor_Game(Saber_PKE_Scheme, A).main() @ &m : res]
  =
  Pr[Delta_Cor_PProg.main() @ &m : res].
```

Listing 1.17. $\Pr\left[\text{Game}^{\text{FOCOR}}_{\mathcal{A},\text{Saber.PKE}} = 1\right] = \Pr\left[\text{PProg}^{\delta\text{COR}} = 1\right]$

Naturally, the veracity of this lemma immediately follows from the previously verified equivalences presented in Listing 1.12 and Listing 1.16 (and their transitivity). This completes the formal verification of Saber.PKE's correctness property. Combining this with the preceding security-related results, we have formally verified that Saber.PKE possesses the properties necessary to transform it into a IND-CCA2 secure and (sufficiently) correct KEM via the relevant variant of the FO transform.

References

ABB+20. José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub.

The last mile: High-assurance and high-speed cryptographic implementations. In *2020 IEEE Symposium on Security and Privacy*, pages 965–982, San Francisco, CA, USA, May 18–21, 2020. IEEE Computer Society Press.

ABH+21. Joël Alwen, Bruno Blanchet, Eduard Hauck, Eike Kiltz, Benjamin Lipp, and Doreen Riepel. Analysing the HPKE standard. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology – EUROCRYPT 2021, Part I*, volume 12696 of *Lecture Notes in Computer Science*, pages 87–116, Zagreb, Croatia, October 17–21, 2021. Springer, Heidelberg, Germany.

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

BBF+21. Manuel Barbosa, Gilles Barthe, Xiong Fan, Benjamin Grégoire, Shih-Han Hung, Jonathan Katz, Pierre-Yves Strub, Xiaodi Wu, and Li Zhou. EasyPQC: Verifying post-quantum cryptography. Cryptology ePrint Archive, Report 2021/1253, 2021.

BCG+12. 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, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

BDF+11. 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, Seoul, South Korea, December 4–8, 2011. Springer, Heidelberg, Germany.

BPR12. 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, Cambridge, UK, April 15–19, 2012. Springer, Heidelberg, Germany.

BR06. Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Serge Vaudenay, editor, *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 409–426, St. Petersburg, Russia, May 28 – June 1, 2006. Springer, Heidelberg, Germany.

CHH+17. Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 1773–1788, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press.

D'A21. Jan-Pieter D'Anvers. *Design and Security Analysis of Lattice-Based Post-Quantum Encryption*. Ph.D. Dissertation, KU Leuven Arenberg Doctoral School, May 2021.

DHK+21. Julien Duman, Kathrin Hövelmanns, Eike Kiltz, Vadim Lyubashevsky, and Gregor Seiler. Faster Kyber and Saber via a generic Fujisaki-Okamoto transform for multi-user security in the QROM. 2021.

DKRV18. 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, Marrakesh, Morocco, May 7–9, 2018. Springer, Heidelberg, Germany.

HHK17. 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, Baltimore, MD, USA, November 12–15, 2017. Springer, Heidelberg, Germany.

KM19. 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.

LCWZ14. 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, 2014.

Mos18. Michele Mosca. Cybersecurity in an era with quantum computers: Will we be ready? *IEEE Security & Privacy*, 16:38–41, September 2018.

Sho94. P.W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994.

Unr20. 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.

Yan13. Song Y. Yan. *Quantum Attacks on Public-Key Cryptosystems*. Springer, Boston, MA, 1st edition, April 2013.

Appendix

A Random Oracle Model Proofs

In the ensuing, we substantiate the hardness of the GMLWR and XMLWR games by means of random oracle model proofs. In essence, these proofs show that if the given gen is a random oracle and the MLWR game is hard, then the GMLWR and XMLWR games (with this gen) are hard. More precisely, assuming the given 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. Akin to the discussions in the paper's main body, we start by presenting the hand-written proof and discuss the corresponding formal verification afterward.

Hand-Written Proof. Foremost, to prevent confusion, we introduce separate identifiers and definitions for the GMLWR and XMLWR games in the random oracle model; in these definitions, to distinguish between a standard gen and an idealized counterpart, we denote the latter by Gen. Figure 12 provides the exact identifiers and definitions.

$\mathrm{GameROM}^{\mathrm{GMLWR}}_{\mathcal{A},\mathsf{Gen},l,\mu,q,p}(u)$

1 : $\mathrm{seed}_{\mathbf{A}} \leftarrow\$\ \mathcal{U}(\{0,1\}^{256})$
2 : $\mathbf{A} \leftarrow \mathsf{Gen}(\mathrm{seed}_{\mathbf{A}})$
3 : $\mathbf{s} \leftarrow\$\ \beta_\mu(R_q^{l\times 1})$
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** $\mathcal{A}^{\mathsf{Gen}}(\mathrm{seed}_{\mathbf{A}}, \mathbf{b}_u)$

$\mathrm{GameROM}^{\mathrm{XMLWR}}_{\mathcal{A},\mathsf{Gen},l,\mu,q,p}(u)$

1 : $\mathrm{seed}_{\mathbf{A}} \leftarrow\$\ \mathcal{U}(\{0,1\}^{256})$
2 : $\mathbf{A} \leftarrow \mathsf{Gen}(\mathrm{seed}_{\mathbf{A}})$
3 : $\mathbf{s} \leftarrow\$\ \beta_\mu(R_q^{l\times 1})$
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** $\mathcal{A}^{\mathsf{Gen}}(\mathrm{seed}_{\mathbf{A}}, \mathbf{b}_u, \mathbf{a}, d_u)$

Figure 12. The GMLWR and XMLWR Games in the Random Oracle Model

Evidently, as desired, $\mathrm{GameROM}^{\mathrm{GMLWR}}_{\mathcal{A},\mathsf{Gen},l,\mu,q,p}(u)$ and $\mathrm{GameROM}^{\mathrm{XMLWR}}_{\mathcal{A},\mathsf{Gen},l,\mu,q,p}(u)$ are almost identical to their standard model analogs, merely substituting Gen for gen (and explicitly providing $\mathcal{A}$ access to Gen).

Concerning the reduction from MLWR to GMLWR, consider an adversary $\mathcal{A}$ against $\mathrm{GameROM}^{\mathrm{GMLWR}}_{\mathcal{A},\mathsf{Gen},l,\mu,q,p}(u)$. Given this adversary, we can straightforwardly construct an adversary $\mathcal{R}^{\mathcal{A}}$ against $\mathrm{Game}^{\mathrm{MLWR}}_{\mathcal{R}^{\mathcal{A}},l,l,\mu,q,p}(u)$ with an advantage

that is equal to the advantage of $\mathcal{A}$ against $\mathrm{GameROM}^{\mathrm{GMLWR}}_{\mathcal{A},\mathsf{Gen},l,\mu,q,p}(u)$. 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 $\mathrm{GameROM}^{\mathrm{GMLWR}}_{\mathcal{A},\mathsf{Gen},l,\mu,q,p}(u)$ is uniformly distributed over $R_q^{l\times l}$, similarly to its counterpart in $\mathrm{Game}^{\mathrm{MLWR}}_{\mathcal{A},l,l,\mu,q,p}(u)$. 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 $\mathrm{Game}^{\mathrm{MLWR}}_{\mathcal{A},l,l,\mu,q,p}(u)$ directly gives $\mathbf{A}$ to its adversary, $\mathrm{GameROM}^{\mathrm{GMLWR}}_{\mathcal{A},\mathsf{Gen},l,\mu,q,p}(u)$ provides the seed with which Gen is queried to obtain $\mathbf{A}$. Combining these observations, we can construct the above-mentioned adversary $\mathcal{R}^{\mathcal{A}}$ against $\mathrm{Game}^{\mathrm{MLWR}}_{\mathcal{R}^{\mathcal{A}},l,l,\mu,q,p}(u)$ as follows.

1. Upon being called by $\mathrm{Game}^{\mathrm{MLWR}}_{\mathcal{R}^{\mathcal{A}},l,l,\mu,q,p}(u)$, $\mathcal{R}^{\mathcal{A}}$ 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 $\mathrm{seed}_{\mathbf{A}} \leftarrow\$\, \mathcal{U}(\{0,1\}^{256})$.
3. Then, $\mathcal{R}^{\mathcal{A}}$ calls $\mathcal{A}(\mathrm{seed}_{\mathbf{A}}, \mathbf{b}_u)$ and proceeds to monitor all random oracle queries. If $\mathcal{A}$ queries the random oracle on $\mathrm{seed}_{\mathbf{A}}$, $\mathcal{R}^{\mathcal{A}}$ 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 $\mathcal{A}(\mathrm{seed}_{\mathbf{A}}, \mathbf{b}_u)$.

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., $\mathrm{GameROM}^{\mathrm{GMLWR}}_{\mathcal{A},\mathsf{Gen},l,\mu,q,p}(u)$, using the values provided by its own game, i.e., $\mathrm{Game}^{\mathrm{MLWR}}_{\mathcal{R}^{\mathcal{A}},l,l,\mu,q,p}(u)$. Hence, the reduction adversary successfully employs $\mathcal{A}$ to obtain an advantage against $\mathrm{Game}^{\mathrm{MLWR}}_{\mathcal{R}^{\mathcal{A}},l,l,\mu,q,p}(u)$ that is equal to the advantage of $\mathcal{A}$ against $\mathrm{GameROM}^{\mathrm{GMLWR}}_{\mathcal{A},\mathsf{Gen},l,\mu,q,p}(u)$.

For the reduction from MLWR to XMLWR, consider an adversary $\mathcal{A}$ against $\mathrm{GameROM}^{\mathrm{XMLWR}}_{\mathcal{A},\mathsf{Gen},l,\mu,q,p}(u)$. Based on this adversary, we construct an adversary $\mathcal{R}^{\mathcal{A}}$ against $\mathrm{Game}^{\mathrm{MLWR}}_{\mathcal{R}^{\mathcal{A}},l+1,l,\mu,q,p}(u)$. In this construction, we utilize the following two observations. First, the i-th entry of the result of a matrix-vector multiplication is equal to the inner product of the matrix's i-th row with the multiplication's operand vector. Second, extracting a row from a uniformly distributed matrix produces a matrix and a vector that (both) are also uniformly distributed. Employing these observations, we can construct adversary $\mathcal{R}^{\mathcal{A}}$ against $\mathrm{Game}^{\mathrm{MLWR}}_{\mathcal{R}^{\mathcal{A}},l+1,l,\mu,q,p}(u)$ as follows.

1. Upon being called by $\mathrm{Game}^{\mathrm{MLWR}}_{\mathcal{R}^{\mathcal{A}},l+1,l,\mu,q,p}(u)$, $\mathcal{R}^{\mathcal{A}}$ 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 $\mathrm{seed}_{\mathbf{A}'} \leftarrow\$\, \mathcal{U}(\{0,1\}^{256})$.

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 transpose 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 and, as such, successfully employs $\mathcal{A}$ to obtain an advantage against $\text{Game}^{\text{MLWR}}_{\mathcal{R}^{\mathcal{A}},l+1,l,\mu,q,p}(u)$ that is equal to the advantage of $\mathcal{A}$ against $\text{GameROM}^{\text{XMLWR}}_{\mathcal{A},\mathsf{Gen},l,\mu,q,p}(u)$.

Formal Verification. Next, we cover the formal verification of the random oracle model proofs. To this end, since both proofs are quite alike, we exclusively consider the formal verification of the proof regarding the reduction from MLWR to GMLWR; the formal verification of the other proof proceeds analogously. As with the discussions in the main body of the paper (and for the same reasons), we do not go over the concrete proofs of the results in EasyCrypt; nevertheless, the code corresponding to the formal verification of (both of) the random oracle model proofs can be found in the repository belonging to this work.

Due to the ubiquity of the random oracle model 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 programmable random oracles. Listing 1.18 provides the relevant part of the module type utilized to formalize such random oracles in EasyCrypt.

```
1  module type PRO = {
2    proc init()
3    proc get(x : in_t) : out_t
4    proc set(x : in_t, y : out_t)
5    (*...*)
6  }.
```

Listing 1.18. 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.

As discussed earlier, given an adversary $\mathcal{A}$ against $\mathrm{GameROM}^{\mathrm{GMLWR}}_{\mathcal{A},\mathsf{Gen},l,\mu,q,p}$, we construct an adversary $\mathcal{R}^{\mathcal{A}}$ against $\mathrm{Game}^{\mathrm{MLWR}}_{\mathcal{R}^{\mathcal{A}},l,l,\mu,q,p}$. Hence, for the formal verification, we foremost require a formalization of these (classes of) adversaries and games. The classes of adversaries are formalized through the module types defined in Listing 1.19.

```
1  module type Adv_MLWR = {
2    proc guess(_A : Rq_mat, b : Rp_vec) : bool
3  }.
4
5  module type Adv_GMLWR_RO(Gen : PRO) = {
6     proc guess(sd : seed, b : Rp_vec) : bool { Gen.get }
7  }.
```

Listing 1.19. Module Types for Adversaries Against $\mathrm{Game}^{\mathrm{MLWR}}_{\mathcal{A},l,l,\mu,q,p}$ and $\mathrm{GameROM}^{\mathrm{GMLWR}}_{\mathcal{A},\mathsf{Gen},l,\mu,q,p}$

As their names suggest, from top to bottom, these module types respectively denote the classes of adversaries against $\mathrm{Game}^{\mathrm{MLWR}}_{\mathcal{A},l,l,\mu,q,p}$ and $\mathrm{GameROM}^{\mathrm{GMLWR}}_{\mathcal{A},\mathsf{Gen},l,\mu,q,p}$. 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 only 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.

Following, we formalize $\mathrm{Game}^{\mathrm{MLWR}}_{\mathcal{A},l,l,\mu,q,p}$ and $\mathrm{GameROM}^{\mathrm{GMLWR}}_{\mathcal{A},\mathsf{Gen},l,\mu,q,p}$ by means of parameterized modules. More precisely, the formalization of $\mathrm{Game}^{\mathrm{MLWR}}_{\mathcal{A},l,l,\mu,q,p}$, `MLWR`, is defined with respect to a parameter of type `Adv_MLWR`; analogously, the formalization of $\mathrm{GameROM}^{\mathrm{GMLWR}}_{\mathcal{A},\mathsf{Gen},l,\mu,q,p}$, `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 way it obtains `_A`; specifically, while `GMLWR` evaluates `gen sd`, `GMLWR_RO` queries the random oracle on `sd`. Certainly, this is consistent with the differences between $\mathrm{GameROM}^{\mathrm{GMLWR}}_{\mathcal{A},\mathsf{Gen},l,\mu,q,p}$ and $\mathrm{Game}^{\mathrm{GMLWR}}_{\mathcal{A},\mathsf{Gen},l,\mu,q,p}$. Due to its vast similarity with `GMLWR`, `GMLWR_RO` is not explicitly presented here. Contrarily, albeit a nearly verbatim translation of $\mathrm{Game}^{\mathrm{MLWR}}_{\mathcal{A},l,l,\mu,q,p}$, `MLWR` is specified in Listing 1.20 for reasons of completeness.

```
1  module MLWR(A : Adv_MLWR) = {
2      proc main(u : bool) : bool = {
3          (*...*)
4          _A <$ dRq_mat;
5          s <$ dsmallRq_vec;
6
7          if (u) {
8              b <$ dRp_vec;
```

```
9           } else {
10              b <- scaleroundRqv2Rpv (_A *^ s);
11          }
12          u' <@ A.guess(_A, b);
13          return u';
14      }
15  }.
```

Listing 1.20. $\text{Game}^{\text{MLWR}}_{\mathcal{A},l,l,\mu,q,p}$

Then, utilizing the above-introduced module types, Listing 1.21 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 $\text{GameROM}^{\text{GMLWR}}_{\mathcal{A},\text{Gen},l,\mu,q,p}$.

```
1   module AGM(AG : Adv_GMLWR_RO) : Adv_MLWR = {
2     module AG = AG(Gen)
3
4     proc guess(_A : Rq_mat, b : Rp_vec) : bool = {
5       (*...*)
6       Gen.init();
7
8       sd <$ dseed;
9
10      Gen.set(sd, _A);
11
12      u' <@ AG.guess(sd, b);
13      return u';
14    }
15  }.
```

Listing 1.21. 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[18]; 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. First, `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 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`, the ma-

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

trix 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`. Certainly, 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, `GMLWR_RO`. Ultimately,`AGM` directly returns the value obtained from the call to `AG.guess(sd, b)`. Finally, we formalize the desired result, i.e., the equality of advantages for $\mathcal{A}$ against $\text{GameROM}^{\text{GMLWR}}_{\mathcal{A},\mathsf{Gen},l,\mu,q,p}$ and $\mathcal{R}^{\mathcal{A}}$ against $\text{Game}^{\text{MLWR}}_{\mathcal{R}^{\mathcal{A}},l,l,\mu,q,p}$. Specifically, we do so through the lemma given in Listing 1.22.

```
1  lemma Equal_Advantage_GMLWR_RO_MLWR &m :
2    `| Pr[GMLWR_RO(A).main(true) @ &m : res] -
3       Pr[GMLWR_RO(A).main(false) @ &m : res] |
4    =
5    `| Pr[MLWR( AGM(A) ).main(true) @ &m : res] -
6       Pr[MLWR( AGM(A) ).main(false) @ &m : res] |.
```

Listing 1.22. Equality of Advantages for $\mathcal{A}$ Against $\text{GameROM}^{\text{GMLWR}}_{\mathcal{A},\mathsf{Gen},l,\mu,q,p}$ and $\mathcal{R}^{\mathcal{A}}$ Against $\text{Game}^{\text{MLWR}}_{\mathcal{R}^{\mathcal{A}},l,l,\mu,q,p}$

Here, `A` denotes an arbitrary module of type `Adv_GMLWR_RO`; that is, `A` formalizes an arbitrary adversary against $\text{GameROM}^{\text{GMLWR}}_{\mathcal{A},\mathsf{Gen},l,\mu,q,p}$. As such, extrapolating from the discussion surrounding Listing 1.7, we can recognize the following correspondences.

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

This shows that `Equal_Advantage_GMLWR_RO_MLWR` accurately formalizes the desired equality of advantages.